



MATLAB an der RWTH

Tutorium

Einführung / Toolboxen

Version 1.7 vom 21. Februar 2011

Inhaltsverzeichnis

1 Einleitung

MATLAB ist eines der am meisten verbreiteten Programme zum wissenschaftlichen, numerischen Rechnen, insbesondere in der Regelungstechnik. Neben den numerischen Funktionalitäten bietet MATLAB auch Möglichkeiten zur graphischen Darstellung und zum Programmieren eigener Skripte. Es handelt sich um ein interaktives System mit Matrizen als Basis-Datenelement. MATLAB steht daher für Matrizen Laboratorium. SIMULINK ist eine auf MATLAB aufgesetzte graphische Benutzeroberfläche, mit der komplexe Systeme modelliert und simuliert werden können. Die notwendigen Matrixberechnungen werden von MATLAB automatisch durchgeführt. Hersteller dieser beiden Produkte ist die Firma The MathWorks Inc. in den USA.

Das Ziel der Übung ist eine erste Einarbeitung in MATLAB/SIMULINK. Bei dieser Übungsanleitung steht weniger die Vollständigkeit der Darstellung als vielmehr die Anleitung zum selbständigen weiteren Arbeiten im Vordergrund. Kleinere Übungen zum besseren Verständnis sind innerhalb der Kapitel integriert. Die mit dem Handsymbol „☞“ markierten Aufgaben am Ende jedes Kapitels dienen der Überprüfung der erworbenen Kenntnisse. Darüber hinaus werden an verschiedenen Stellen Vorschläge zum selbständigen Experimentieren gemacht.

Bitte beachten Sie: Diese Anleitung ist auf dem Stand von MATLAB **Version R2008b**. Sollten Sie mit einer neueren Version arbeiten, können sich insbesondere Menüstrukturen und Dialogboxinhalte verändert haben. Sollten Sie die beschriebenen Befehle oder Funktionen nach etwas Suchen nicht finden können, wenden Sie sich bitte an den Betreuer.

2 Installation und Borrowing

2.1 Installation von Matlab (Windows)

Zur Installation von MATLAB laden Sie die Installationsdateien vom Rechenzentrum herunter. Weitere Hinweise dazu finden Sie auf der Internetseite www.matlab.rwth-aachen.de. Diese Anleitung bezieht sich auf die Installation unter Windows. Hinweise zur Installation unter anderen Betriebssystemen finden Sie in den entsprechenden Ordnern im Downloadverzeichnis.

Wenn Sie die Software schon vom Rechenzentrum heruntergeladen bzw. ein Windows-Netzwerklaufwerk eingerichtet haben (lesen Sie dazu die E-Mail, die Sie nach dem Erwerb von Asknet bekommen), müssten Sie eine ähnliche Ordnerstruktur wie in Abbildung 2.1 vorfinden.

In der Datei `.\RWTH-Readme.txt` befindet sich eine kurze Anleitung zum Installationsverfahren und der Schlüssel für die Installation.

- Führen Sie die Datei `.\net-install\win\setup.exe` aus, um die Installation zu starten.
- Wählen Sie **Install manually without using the internet** aus.
- Im Dialogfenster **File Installation Key** verwenden Sie den Key aus `.\RWTH-Readme.txt`.
- Im Dialogfenster **Choose installation type** wählen Sie **Custom** aus und folgen den weiteren Anweisungen bis zum Dialogfenster **Product Selection**.

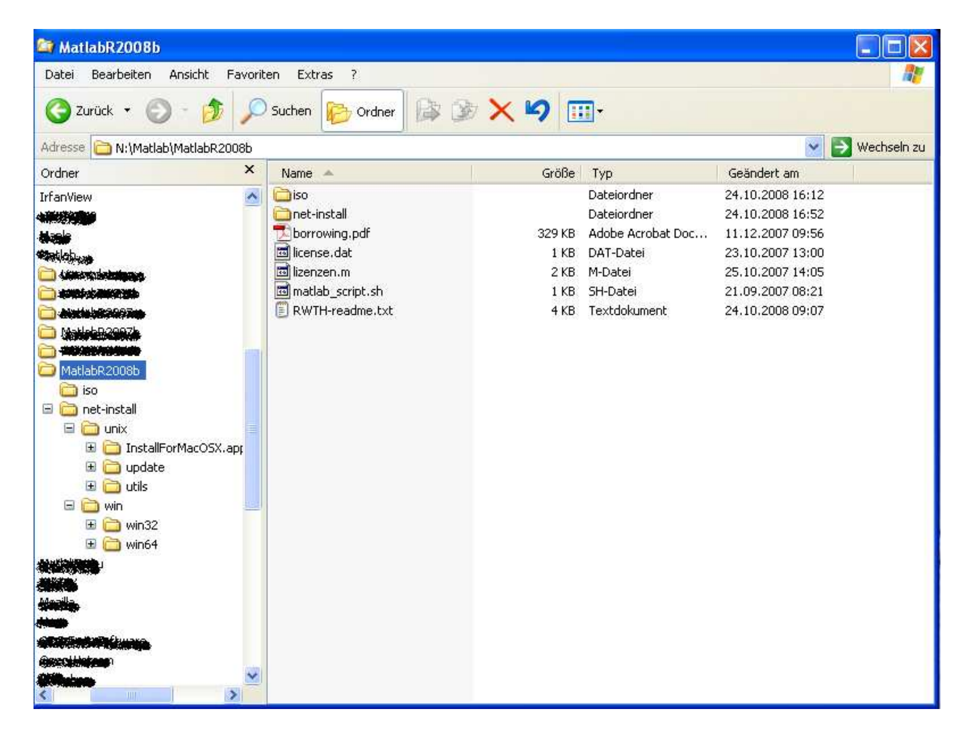


Abbildung 2.1: Ordnerstruktur

- Im Dialogfenster **Product Selection** (Abbildung 2.2) können Sie den **License Manager** mitinstallieren, falls Sie später das Borrowing verwenden möchten. Damit können Sie bis zu 30 Tage Matlab ohne eine Verbindung zum Lizenz-Server am Rechenzentrum der RWTH Aachen einsetzen (näheres dazu im nächsten Abschnitt).
- Im Dialogfenster **License File** wählen Sie die `.\license.dat` Datei aus.
- Falls Sie den **License Manager** hinzugefügt haben, erscheint das Dialogfenster **License Manager Configuration**. Hier wählen Sie **Configure license manager service** aus und folgen den weiteren Anweisungen.
- Nach der Installation kopieren Sie die `.\license.dat` Datei in das Verzeichnis `C:\Programme\MATLAB\R2008b\licenses\`.

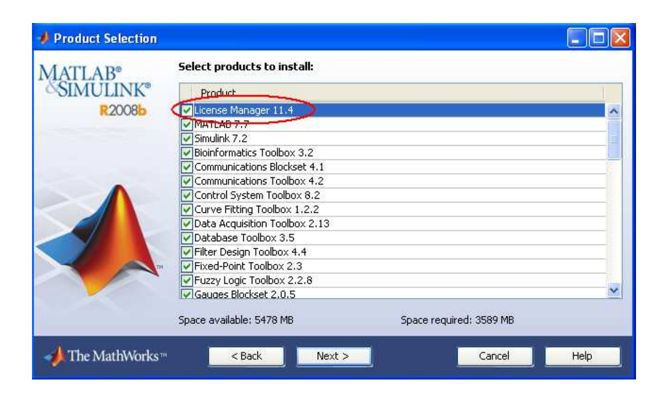


Abbildung 2.2: Product Selection

Somit ist die Installation abgeschlossen. Falls beim Starten von Matlab ein **Activation Welcome**-Fenster angezeigt wird, überprüfen Sie bitte den Inhalt der `license.dat` Datei gemäß der Anleitung in `.\RWTH-Readme.txt`.

Die knappe Darstellung der hier beschriebene Schritte ersetzt nicht die Installationsanleitung. Falls Probleme bei der Installation auftreten sollten, lesen Sie bitte zunächst die vollständige Anleitung in `.\net-install\win\install_guide.pdf`. Wenn das Problem nicht behoben werden kann, wenden Sie sich bitte an den auf der Homepage www.matlab.rwth-aachen.de angegebenen Ansprechpartner.

2.2 Borrowing von Lizenzen

Die Lizensierung von Matlab erfolgt standardmäßig über den Lizenz-Server des Rechen- und Kommunikationszentrums, sodass für den Einsatz der Software eine Verbindung mit dem Netz der RWTH Aachen bestehen muss (z.B. über VPN). Das Borrowingverfahren ermöglicht eine Offline-Nutzung von Matlab. Somit können die Lizenzen der verschiedenen Toolboxen **bis zu 30 Tage** ausgeliehen werden.

Die Vorgehensweise ist vom Betriebssystem und der verwendeten Matlab-Version abhängig. Hier werden zwei Möglichkeiten des License Borrowings vorgestellt. Eine Möglichkeit liegt in der Verwendung des LMTOOLS. Dies funktioniert nur mit Windows, ist jedoch unabhängig von der verwendeten Matlab-Version. Wenn Sie bereits Matlab R2009b verwenden, haben Sie zusätzlich die Möglichkeit, die License Borrowing UI von Matlab zu verwenden (unabhängig vom Betriebssystem).

Falls Sie eine ältere MATLAB-Version in Kombination mit einem anderen Betriebssystem als Windows verwenden, können Sie eine entsprechende Dokumentation in `.\borrowing.pdf` finden. Diese Datei befindet sich auch auf der Homepage der Campuslizenz www.matlab.rwth-aachen.de unter dem Namen **Guide to License Borrowing**.

Borrowing mit LMTOOLS (nur Windows)

LMTOOLS ist ein graphisches User Interface zum License-Management und kann verwendet werden, um das Borrowing einzurichten.

Installation von LMTOOLS

Wenn Sie Matlab noch nicht installiert haben oder auf die neueste Matlab Version updaten, können Sie **LMTOOLS** (License Manager) direkt bei der Matlab Installation auswählen und mitinstallieren (siehe Abbildung 2.2). Dieses Tool wird in der Standardinstallation nicht installiert.

Falls Sie aber Matlab schon installiert haben und das Tool nachinstallieren möchten, dann müssen Sie **nur** License Manager auswählen, sodass nur dieses Tool hinzugefügt wird und nicht die anderen Pakete installiert werden müssen (siehe Abbildung 2.3).

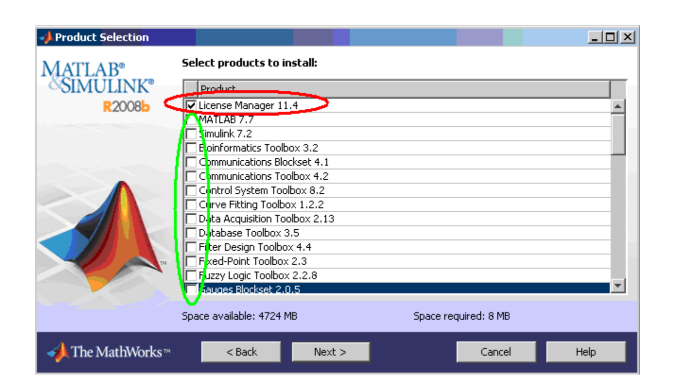


Abbildung 2.3: Nachinstallation von LMTOOLS

Wo befindet sich LMTOOLS?

Sie finden das Tool in `\$MATLAB\flexlm` oder in `\$MATLAB\bin\win32`, wobei `\$MATLAB` für das gewählte Installationsverzeichnis steht. (in der Standardeinstellung z.B. `C:\Programme\MATLAB\R2008b`)

Wie richte ich das Borrowingverfahren ein?

Sie müssen eine Verbindung zum Netz der RWTH haben, um das Verfahren durchführen zu können.

1. Führen Sie das Programm **lmtools.exe** aus, das im oben genannten Verzeichnis liegt.
2. Wählen Sie den **Borrowing** Tab im LMTOOLS Dialogfeld aus.

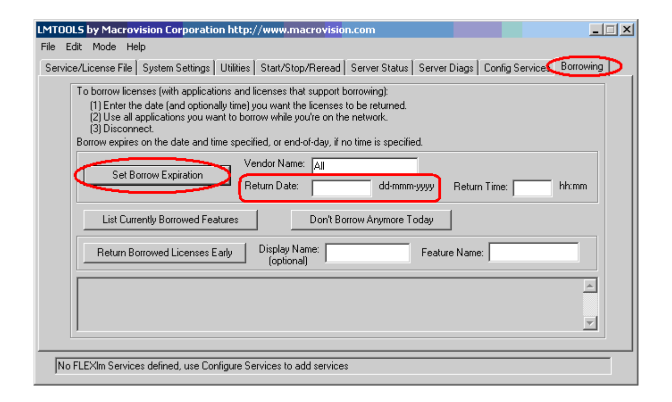


Abbildung 2.4: Einrichten von Borrowing

3. Füllen Sie das Feld für das Rückgabedatum aus. Achten Sie auf das Datumsformat **TT-MMM-JJJJ**. Sie müssen die ersten drei Buchstaben des Monats **auf Englisch** eingeben. Beispielsweise `12-dec-2008`.
4. Klicken Sie auf **Set Borrow Expiration**, um das Borrowing zu initialisieren.
5. Schließen Sie das Fenster noch nicht, um das erfolgreiche Ausleihen der Lizenzen später kontrollieren zu können.

Wie leihe ich die Lizenzen der Toolboxen aus?

Im Prinzip müssen die Toolboxen einmal genutzt werden, während die Verbindung zum Rechenzentrum noch vorhanden ist, um die entsprechende Lizenzen auszuleihen.

Um diese Aufgabe zu erleichtern und alle Produkte der Campuslizenz auf einmal auszuleihen, wurde vom Rechenzentrum ein kleines m-Skript geschrieben. Dieses Skript heißt `lizenzen.m` und befindet sich in demselben Ordner, in dem die MATLAB Installationsdateien liegen. Den Inhalt dieser Datei können Sie aber auch von unten kopieren und als Textdatei mit Endung **.m** speichern.

```
%% Inhalt von lizenzen.m

license checkout MATLAB;
license checkout Communication\_Blocks;
license checkout Curve\_Fitting_Toolbox;
license checkout Communication_Toolbox;
license checkout Compiler;
license checkout Control_Toolbox;
license checkout Data_Acq_Toolbox;
license checkout Database_Toolbox;
license checkout Distrib_Computing_Toolbox;
license checkout Signal_Blocks;
license checkout RTW_Embedded_Coder;
license checkout Excel_Link;
license checkout Fuzzy_Toolbox;
license checkout Fixed-Point_Blocks;
license checkout GADS_Toolbox;
license checkout Instr_Control_Toolbox;
license checkout Identification_Toolbox;
license checkout Image_Toolbox;
license checkout SimDriveline;
license checkout MAP_Toolbox;
license checkout MPC_Toolbox;
license checkout SimMechanics;
license checkout NCD_Toolbox;
license checkout Neural_Network_Toolbox;
license checkout SIMULINK;
license checkout Optimization_Toolbox;
license checkout PDE_Toolbox;
license checkout Fixed_Point_Toolbox;
license checkout RF_Blockset;
license checkout Robust_Toolbox;
license checkout RF_Toolbox;
license checkout Real-Time_Workshop;
license checkout Stateflow_Coder;
license checkout Simulink_Control_Design;
license checkout SimEvents;
license checkout Stateflow;
license checkout Signal_Toolbox;
license checkout Spline_Toolbox;
license checkout Simscape;
license checkout Statistics_Toolbox;
license checkout Embedded_Target_MPC555;
license checkout Virtual_Reality_Toolbox;
license checkout Wavelet_Toolbox;
license checkout XPC_Target;
quit;
```

1. Starten Sie MATLAB und gehen Sie, z.B. mit Hilfe des Knopfes **Browse for folder**, zu dem Verzeichnis, in dem sich `lizenzen.m` befindet (siehe Abbildung 2.5).
2. Schreiben Sie im Command Window `lizenzen` und drücken Sie **Enter**. Alternativ können Sie das Skript durch einen Doppelklick auf die Datei im Current Directory Fenster ausführen.

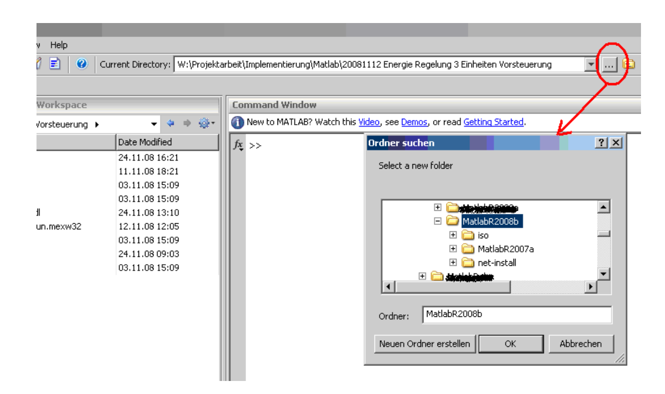


Abbildung 2.5: Pfad wechseln (wo sich lizenzen.m befindet)

3. Dadurch werden alle Lizenzen der Campuslizenz ausgeliehen und MATLAB beendet.

Wie kontrolliere ich die ausgeliehenen Lizenzen und das Rückgabedatum?

Sie können anhand vom LMTOOLS die bereits ausgeliehenen Lizenzen überprüfen und das Rückgabedatum ansehen.

1. Falls notwendig, starten Sie das Programm LMTOOLS.

2. Klicken Sie auf den Knopf **List Currently Borrowed Features**. In dem Fenster werden dann die ausgeliehenen Produkte aufgelistet.

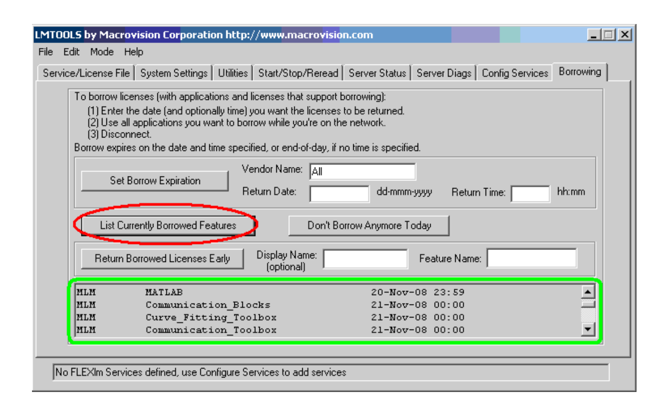


Abbildung 2.6: Kontrollieren der ausgeliehenen Produkte

Wie nutze ich jetzt die ausgeliehenen Lizenzen?

Sie können jetzt MATLAB offline starten und alle Produkte für den eingegebenen Zeitraum benutzen. Sollten Sie Probleme beim Starten von MATLAB haben, kann das daran liegen, dass Sie noch zum Server verbunden sind. Trennen Sie kurz Ihren Rechner vom Netz, um das Problem zu beheben. Mehr Informationen dazu finden Sie in den FAQ auf der Webseite der Campuslizenz.

Was kann man noch mit LMTOOLS machen?

Es besteht beispielsweise die Möglichkeit, ausgeliehene Lizenzen früher zurückzugeben. Darüber hinaus kann das Borrowing abgeschaltet werden, falls Sie nur einige Produkte ausleihen möchten und verhindert werden soll, dass weitere Produkte unabsichtlich ausgeliehen werden.

Weitere Informationen zu diesen Themen können Sie in der Dokumentation **Guide to License Borrowing** finden.

Borrowing über die License Borrowing UI (ab Matlab R2009b)

Die License Borrowing UI ist eine in MATLAB R2009b integrierte graphische Benutzeroberfläche, die das Ausleihen von Lizenzen vereinfacht. Wenn Sie die License Borrowing UI verwenden möchten, müssen Sie sie einmalig aktivieren. Kopieren Sie dafür folgende Zeile in das Command Window und bestätigen Sie mit der Eingabetaste:

```
com.mathworks.mde.licensing.borrowing.BorrowUI.getInstance().enableFeature(true)
```

Starten Sie nun MATLAB neu und wählen Sie im Menü *Help* → *Licensing* → *Borrow Products*. Es erscheint das Fenster „MathWorks License Borrowing" (Abb. 2.7).

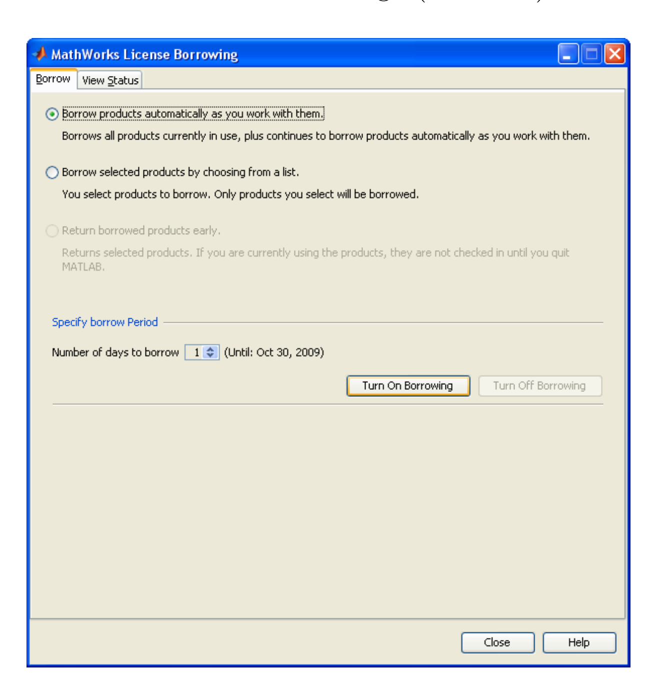


Abbildung 2.7: Automatisches Ausleihen

Hier haben Sie zwei Möglichkeiten.

Automatisches Ausleihen

Wenn Sie einfach die Produkte, die Sie gerade benutzen (oder noch benutzen werden), ausleihen wollen, ohne Sie ausdrücklich anzugeben, wählen Sie „Borrow Products automatically as you work with them.". Geben Sie unter „Number of days to borrow" an, für wieviele Tage (max. 30) Sie die

benutzten Produkte ausleihen wollen, und klicken Sie anschließend auf „Turn Borrowing On“. Sie können das Fenster nun schließen.

Selektives Ausleihen

Sie können aber auch die Produkte, die Sie ausleihen möchten, aus einer Liste auswählen. Markieren Sie dazu den Punkt „Borrow selected products by choosing from a list.“ (Abb. 2.8). Wählen Sie die Produkte, die Sie ausleihen möchten, aus der Liste aus und geben Sie unter „Number of days to borrow“ an, für wieviele Tage (max. 30) Sie sie ausleihen wollen. Produkte, die nicht ausgeliehen werden können, sind in der Liste mit dem *Borrow Status* „Borrowing Disabled“ gekennzeichnet. Schließen Sie den Vorgang mit einem Klick auf den Button „Borrow“ ab. Sie können das Fenster nun schließen.

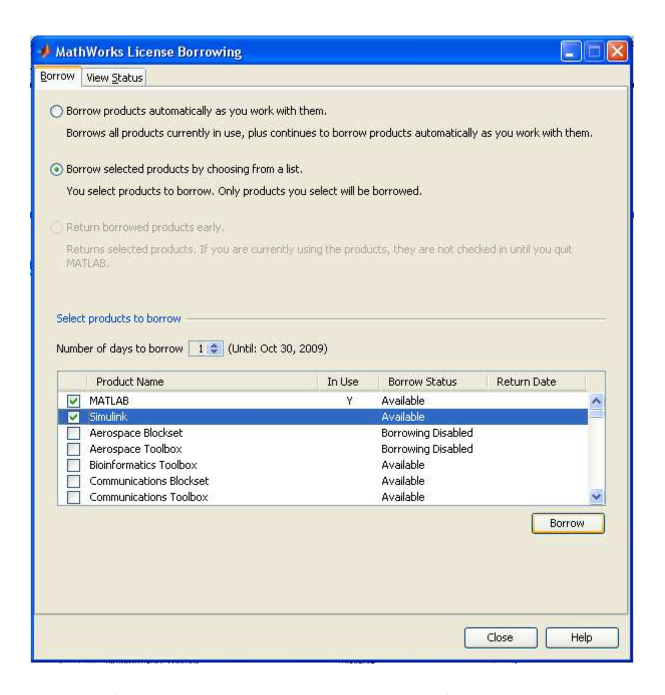


Abbildung 2.8: Selektives Ausleihen

Ausleih-Status

Wenn Sie den Tab „View Status“ auswählen, können Sie nachsehen, welche Lizenzen Sie ausgeliehen haben und wann diese wieder zurückgegeben werden.

3 Einführung MATLAB

3.1 Benutzeroberfläche und Kommandozeile

Befehlsfenster (Kommandozeile)

Nach dem Starten von MATLAB erscheint die MATLAB-Oberfläche (siehe Abb. 3.1). Sie besteht aus mehreren Fenstern, wobei das Befehlsfenster bzw. **Command Window (CW)** mit dem MATLAB-Prompt „>>“ den größten Platz einnimmt. Das Befehlsfenster dient zur Eingabe von Variablen und dem Ausführen von Befehlen, Funktionen und Skripten. Jede Befehlszeile wird mit einem Return ↵ abgeschlossen und der Ausdruck direkt nach der Eingabe ausgewertet. In diesen Unterlagen enthaltene Beispiele beginnen mit dem oben angegebenen Prompt. Kommentare, denen ein Prozentzeichen % vorangestellt ist, brauchen nicht eingegeben zu werden, sondern dienen nur dem Verständnis.

Versuchen Sie, die folgenden Beispiele nachzuvollziehen:

```
>> 5*7          % Zuweisung der Standard-Variablen ans
>> a=3*pi       % Zuweisung der Variablen a
>> b = a/5
>> c = a*b;
>> b = 1:3:20
>> plot(b)
>> help cos
>> b = 2cos(b)
```

Die letzte Zeile verursacht eine der häufigsten Fehlermeldungen: `missing operator, comma, or semi-colon`. In diesem Fall fehlt das ∗-Zeichen zur Multiplikation zwischen 2 und dem Kosinus. Der korrekte Befehl lautet:

```
>> b = 2*cos(b)
```

Die Ausgabe auf dem Bildschirm kann mit einem Semikolon am Ende der Zeile unterdrückt werden. Testen Sie dieses, indem Sie den Befehl mit einem Semikolon wiederholen.

Anstatt den Befehl neu einzugeben, kann auch die Taste ↑ (Pfeil nach oben) gedrückt werden, mit der eingegebene Befehle wiederholt werden. Diese Befehle können vor Eingabe des Return ↵ beliebig verändert werden. Noch schneller geht es, wenn der oder die ersten Buchstaben des Befehls bekannt sind. Gibt man diese ein und drückt dann die Taste ↑, werden nur diejenigen Befehle mit demselben Beginn wiederholt.

Symbolleiste des Befehlsfensters

Die Symbolleiste (vgl. Abb. 3.2) ermöglicht den schnellen Zugriff auf häufig benötigte Funktionen. Die Bedeutung der einzelnen (wichtigeren) Symbole werden von links nach rechts erklärt:

1. Starten des MATLAB-Editors in einem neuen Fenster.
   Dieser Vorgang ist analog zu der Menü-Auswahl *File:New:M.File*. Der Editor und das Erstellen bzw. Verändern eigener Skripte wird später erläutert.

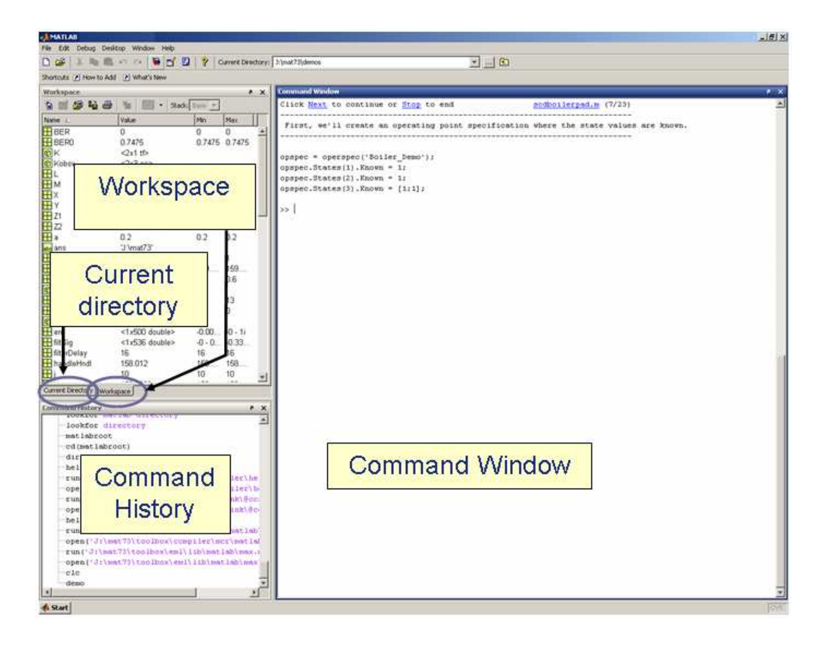


Abbildung 3.1: MATLAB-Oberfläche

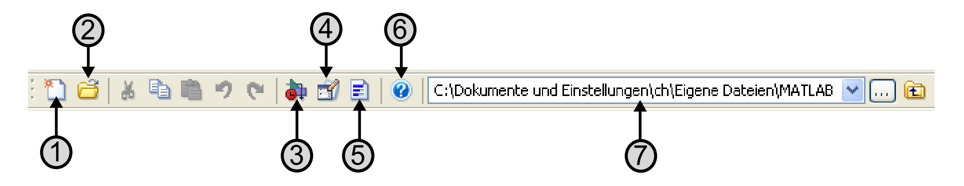


Abbildung 3.2: Symbolleiste des Befehlsfensters

2. Öffnen eines existierenden MATLAB-Dokuments im MATLAB-Editor
   Es wird eine Auswahl mit der Endung `.m` für MATLAB-Skripte, `.mdl` für SIMULINK-Modelle oder `.fig` für MATLAB-Bilder angezeigt. Grundsätzlich kann aber auch jede andere ASCII-Datei im Editor bearbeitet werden.

3. Starten von SIMULINK
   Zur graphischen Modellierung von Systemen in Form gerichteter Graphen. Äquivalent zur Eingabe von `simulink` im Befehlsfenster.

4. Starten von GUIDE
   Ein GUI Editor zur Programmierung von *Graphical User Interfaces*. Äquivalent zur Eingabe von `guide` im Befehlsfenster.

5. Starten von Profiler
   Zur Optimierung von M-Files.

6. Öffnen des Hilfefensters
   Dieser Vorgang ist analog zur Eingabe des Befehls `doc` im Befehlsfenster. Das Hilfefenster enthält einen vollständigen Hilfetext zu einer beliebigen Funktion *fname* im Vergleich zu einer Kurzhilfe bei Eingabe von `help` *fname*. Der Text wird aber nicht im Befehls- sondern in einem gesonderten Fenster, welches auch ein einfacheres Navigieren zwischen den Befehlen erlaubt, angezeigt.

7. Aktuelles Verzeichnis
   In diesem Verzeichnis wird als erstes nach benötigten Dateien gesucht und hier werden zu speichernde Dateien abgelegt. Ist eine Suche im aktuellen Verzeichnis nicht erfolgreich, wird sie dem Suchpfad folgend fortgesetzt. Der Suchpfad kann mit dem Menüpunkt *File:Set Path* festgelegt werden.
   In diesem Zusammenhang gibt es noch ein paar nützliche Befehle im Befehlsfenster:

```
>> cd        % Zeigt den aktuellen Pfad
>> cd ..     % Wechsel in das übergeordnete Verzeichnis
>> cd work   % Wechsel in das untergeordnete Verzeichnis work
>> dir       % Listet alle Dateien im aktuellen Verzeichnis auf
>> what      % Listet alle Matlab-Dateien im Verzeichnis auf
```

Weitere Fenster

In der Abbildung 3.1 sind außerdem drei weitere wichtige Fenster dargestellt. Diese sollen hier kurz beschrieben werden.

Standardmäßig werden alle verwendeten Variablen automatisch im sogenannten (*base*) **Workspace** gespeichert. Im gleichnamigen Fenster werden Größe und Typ der verwendeten Variablen angezeigt. Durch Doppelklick auf die Variablen lassen sich deren Inhalt und Eigenschaften anzeigen und verändern.

Das Fenster **Current Directory** zeigt den Inhalt des aktuellen Verzeichnisses in der üblichen Baumstruktur an. Es sind auch Dateioperationen wie Öffnen, Löschen, Umbenennen und Kopieren in gewohnter Weise möglich.

Im Fenster **Command History** wird eine Liste der zuletzt eingegebenen Befehle angezeigt. Durch Doppelklick auf einen Befehl wird dieser erneut ausgeführt.

Weitere Funktionen der Matlab-Oberfläche

Hier werden einige der verfügbaren Funktionen zur Bedienung der MATLAB-Oberfläche behandelt, die erfahrungsgemäß oft genutzt werden und die Arbeit erleichtern können.

Shortcuts

Mit MATLAB-Shortcuts kann man Befehle zusammenfassen und durch eine Mausaktion später auf einen Schlag durchführen. Zum Editieren des Shortcuts wird ein so genannter Shortcut-Editor verwendet, der über den **Start**-Button (unten links) im Menü Shortcuts aufgerufen werden kann. Diese Shortcuts können entweder im *Start*-Knopf oder als Icon in die **Shortcut-Liste** (oben links) platziert werden

Tab-Komplettierung

Mit Hilfe der Tabulator-Taste können im Command Window Funktionen-, Variablen- und Dateinamen schnell eingegeben werden, indem man nach dem Eintippen des ersten Teils des Wortes die Tabulator-Taste drückt. Der Rest des Namens wird dann automatisch komplettiert oder

ggf. eine Liste passender Möglichkeiten zur Auswahl angezeigt, falls mehrere Namen den gleichen Anfang haben. Dazu muss diese Funktionalität aktiviert werden, was im Menü unter *File:Preferences:Keyboard:Tab completation* möglich ist.

Aufgaben und Übungen

3.1 ☞ *Erste Schritte mit der Kommandozeile:* Geben Sie in der Kommandozeile folgende Befehle ein und versuchen Sie die Ergebnisse nachzuvollziehen.

```
>> sin(pi/4)          % Trigonometrische Funktionen
>> sqrt(2^2+3^2)      % Quadratische Wurzel
>> sqrt(-1)           % Die Imaginärzahl i
>> (5+3*i)*(2+i)      % Rechnen mit komplexen Zahlen
```

3.2 ☞ *RC-Schaltung und komplexe Zahlen:* Es ist die RC-Schaltung aus der Abbildung 3.3 gegeben. Mit Hilfe komplexer Zahlen berechnen Sie die komplexe Spannung am Kondensator. Berechnen Sie den Betrag (Scheitelwert), den Effektivwert und die Phasenverschiebung der Kondensatorspannung gegenüber der Spannung der Quelle in Grad. *Hinweis:* Der Zusammenhang zw. Effektivwert und Scheitelwert ist durch die Gleichung $V_{eff} = \frac{V_{max}}{\sqrt{2}}$ gegeben.

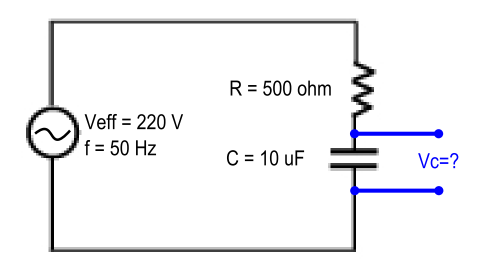


Abbildung 3.3: RC-Schaltung zur Aufgabe 3.2

3.3 ☞ *Verdopplung der Frequenz:* Untersuchen Sie die Wirkung einer Verdopplung der Frequenz in Aufgabe 3.2. Ist die Kondensatorspannung betragsmäßig größer oder kleiner? Warum?

3.2 Die Matlab-Hilfe

Meistens wird die direkte Hilfe zu einer Funktion, insbesondere zu ihrer Syntax benötigt. Dazu gibt man den Befehl `help` (oder `doc` für mehr Details) gefolgt vom Funktionsnamen ein:

```
>> help cos
>> help linspace
>> doc linspace
>> lookfor cosine
```

Der Befehl `lookfor` durchsucht in den Dateien im Suchpfad alle ersten Zeilen des Hilfetexts nach dem angegebenen Wort.

Über die Eingabe von Befehlen lassen sich weitere Hilfemöglichkeiten aufrufen:

- Help Window
  Eigenes Hilfefenster zum Anzeigen des Hilfetexts einer Funktion (s. Symbolleiste). Dazu kann durch Doppelklicken auf einen Eintrag in der Gliederung **Help Topics** ein Bereich durchsucht werden. Das Hilfefenster erscheint ebenfalls durch Eingabe des Befehls `helpwin` im Befehlsfenster.

- Help Desk
  Das Help Desk enthält in Form von `.html` oder `.pdf` Dateien ausführlichere Informationen zu den einzelnen Funktionen. Das Help Desk erscheint ebenfalls durch Eingabe des Befehls `helpdesk` im Befehlsfenster. Dieser Befehl ist äquivalent zu der Eingabe von `doc` ohne einen Funktionsnamen.

- Examples und Demos
  Es erscheint das Demo Fenster mit einer Liste von mitgelieferten Beispielen und Demonstrationen. Das Demo Fenster erscheint ebenfalls durch Eingabe von `demo` im Befehlsfenster.

3.4 ☞ *Suche in der Hilfe:* Ermitteln Sie mit Hilfe von `lookfor` den Befehl, der den Tangens Hyperbolicus (hyperbolic tangent) berechnet. Berechnen Sie dann den Tangens Hyperbolicus von 0,5.

3.5 ☞ *Dreieck im CW:* Berechnen Sie die Fläche und den Umfang eines gleichschenkligen Dreiecks mit Höhe $h = 10$ cm und Breite (Basis) $b = 5$ cm im Befehlsfenster.

3.6 ☞ *Nutzen von help:* Ermitteln Sie die Bedeutung der Befehle `which`, `pause` und `disp`.

3.3 Skripte und der Matlab-Editor

Neben dem Befehlsfenster am MATLAB-Prompt können Befehlsfolgen auch in **Skripte** eingetippt werden, sodass diese auf einmal der Reihe nach ausgeführt werden können. Die Skript-Textdateien haben die Endung `.m` und werden deswegen auch **M-Files** genannt.

Bei seinem Aufruf stehen dem Skript-File alle im *(base)* Workspace gespeicherten Variablen zur Verfügung. Umgekehrt sind alle Variablen, die im Skript-File definiert werden, nach dessen Ablauf noch im Workspace definiert (d.h., die Variablen des Skript-Files sind **global**). Das ist ein wesentlicher Unterschied zu den m-Funktionen (behandelt in 3.9), die einen eigenen Workspace haben und keinen Zugriff auf den *(base)* Workspace.

Obwohl diese Dateien in einem beliebigen Texteditor bearbeitet werden können, bietet MATLAB den so genannten MATLAB-Editor an, der neben einer Färbung der Syntax die Möglichkeit bietet, das Skript durch Mausaktionen auszuführen und zu debuggen. Hier kann man nicht nur den Code Schritt für Schritt ausführen, sondern auch die Variablenwerte anzeigen lassen. Der MATLAB-Editor (siehe Abb. 3.4) kann über das Menü *File* oder durch Eingabe des Befehls `edit` aufgerufen werden.

Wenn Skripte zu lang werden, kann man sinnvollerweise den sogenannten **Cell Mode** verwenden. Das Skript wird damit durch **Cell Dividers %%** in Abschnitte unterteilt. Der *Cell Mode* ermöglicht darüber hinaus eine schnelle Navigation durch das Skript. Um von einer Cell zu den nächsten oder den vorherigen zu wechseln, benutzt man die Tastenkombinationen *Strg*+ ↓ oder *Strg*+ ↑.

Abbildung 3.4 zeigt ein Screenshot des Editorfensters. Die wichtigsten Elemente sind markiert und werden im Folgenden erläutert.

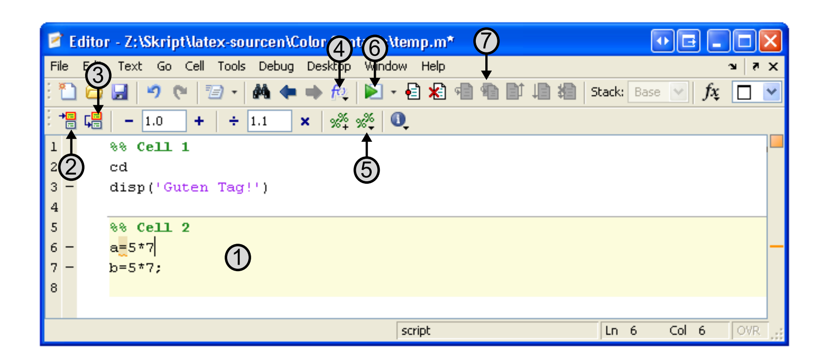


Abbildung 3.4: Das MATLAB-Editorfenster

1. Beispiel einer Cell. Der Editor zeigt durch Farben und Linien die Grenzen der aktuellen Cell an.

2. Der aktuelle Code-Abschnitt wird ausgeführt.

3. Der aktuelle Code-Abschnitt wird ausführt und zum nächsten gewechselt.

4. Durch diesen Knopf lassen sich die Funktionen (später behandelt) der aktuellen Datei schnell erreichen.

5. Damit ist es möglich zu einem beliebigen Abschnitt in der Datei zu gelangen. Es erscheint hier eine Liste mit den Beschriftungen der vorhandenen Abschnitte.

6. Mit diesem Knopf lässt sich das komplette Skript ausführen. Dasselbe kann durch Drücken der Taste *F5* erreicht werden.

7. Dieser Bereich ist für das Debuggen von Bedeutung. Damit kann beispielsweise das Skript schrittweise ausgeführt werden.

Der Editor bietet auch andere nützliche Funktionalitäten an, wie das **Ein-** und **Auskommentieren** mehrerer Codezeilen. Dies erfolgt über das Menü *Text:Comment* und *Text:Uncomment*. Von Bedeutung ist außerdem die Möglichkeit, den Code mit Hilfe vom Menü *Text:Smart Indent* automatisch einzudrücken, sodass dieser übersichtlicher wird.

Aufgaben und Übungen

3.7 ☞ *Dreieck als Skript:* Portieren Sie den Code aus der Aufgabe 3.5 in ein Skript, damit er mit einem Mausklick ausgeführt werden kann. Mit Hilfe der Befehle `input` und `disp` gestalten Sie Ihr Skript so, dass es interaktiv nach der Höhe und der Breite fragt.

3.8 ☞ *Steinfall:* Ein Stein fällt aus einer unbekannten Höhe zu Boden. Die Fallzeit wird mit einer Uhr gemessen und beträgt 10 sec. Schreiben Sie ein Skript, das die Höhe ermittelt, wenn davon ausgegangen werden kann, dass der Luftwiderstand vernachlässigbar ist.

3.9 ☞ *Der Matlab-Editor:* Öffnen Sie eine neue Datei im Editor und tragen Sie die Befehle

```
% Das ist der erste Versuch!
cd
disp('Guten Tag!')
a=5*7
b=5*7;
```

ein. Speichern Sie die Datei in Ihrem Arbeitsverzeichnis unter dem Namen `versuch1.m`. Schließen Sie den Editor. Geben Sie im Befehlsfenster den neuen Befehl `versuch1` und `help versuch1` ein. Was macht der neue Befehl?

Wechseln Sie in das Fenster *Current Directory*. Öffnen Sie dort Ihre Datei `versuch1.m` durch Doppelklick. Drücken Sie im Editor die Taste *F5*. (Damit speichern Sie die Datei und bringen sie sofort zur Ausführung.)

Passen Sie das Skript wie in der Abb. 3.4 an und probieren Sie die Cell-Knöpfe 1 und 2 aus.

3.10 ☞ *Shortcut Loeschen:* Schreiben Sie ein Skript, das mit dem Befehl `clc` den Text im Kommando-Fenster löscht und außerdem die Nachricht *Kommando-Fenster gelöscht!* anzeigt. Erstellen Sie jetzt einen *Shortcut* namens „Loeschen" mit derselben Funktionalität.

3.4 Vektoren und Matrizen

MATLAB wurde speziell zur Berechnung von Matrizen geschrieben. Eine Matrix ist für MATLAB ein Feld mit numerischen Einträgen. Wörter (**Strings**) sind **Zeichenketten**, bei denen die Feldelemente den ASCII-Code (ganzzahlige, positive Werte) der Buchstaben enthalten. MATLAB kennt zwar auch noch andere Wege, alphanumerische Daten zu speichern, jedoch bleiben wir vorerst bei der Vorstellung von numerischen Matrizen.

Konstanten (Skalare) sind intern 1×1-Matrizen. Vektoren sind $1\times n$-Matrizen (Zeilenvektoren) oder $n \times 1$-Matrizen (Spaltenvektoren). Bei Matrix-Berechnungen (z. B. Multiplikation) muss immer genau der Überblick behalten werden, was berechnet werden soll und welche Art von Vektoren dazu verwendet werden. Sonst erhält man als Ergebnis nur Fehlermeldungen oder nicht beabsichtigte Ergebnisse, da etwas anderes berechnet worden ist!

Matrizen mit mehr als einem Element werden zeilenweise eingegeben und mit eckigen Klammern [ ] begrenzt. Innerhalb einer Matrix dienen Kommata oder Leerschritte zur Spaltentrennung und Semikola zur Zeilentrennung:

```
>> [1,2,3]                % Zeilenvektor
>> [4;5;6]                % Spaltenvektor
>> [1,2,3;4,5,6;7,8,0]    % Matrix
>> 'test'                 % Zeichenkette
```

Vektoren können auch schneller in den folgenden Formen eingegeben werden, wenn eine feste (oder logarithmische) Schrittweite verwendet werden soll:

- Erzeugen eines Zeilenvektors mit fester (eingegebenen) Schrittweite:

  `>> [1.Element : Schrittweite : letztes Element]`

- Erzeugen eines Zeilenvektors zwischen zwei Werten mit gleichmäßiger Verteilung:

  `>> linspace(1.Element,letztes Element,Anzahl von Elementen)`

- Erzeugen eines Zeilenvektors zwischen 10^{x1} und 10^{x2} mit logarithmischer Verteilung:

```
>> logspace(x1,x2,Anzahl von Elementen)
```

Aufgaben und Übungen

3.11 ☞ *Erzeugen von Vektoren:* Probieren Sie folgende Befehle aus und versuchen Sie, die Ergebnisse nachzuvollziehen.

```
>> [1:.1:5]
>> 1:10
>> linspace(0,10,5)
>> logspace(-1,2,4)
```

3.12 ☞ *Gleich verteilte Zahlenfolge:* Erzeugen Sie mit Hilfe des Befehls `linspace` eine Zahlenfolge mit 22 Werten zwischen 1,5 und 9.

3.13 ☞ *Logarithmisch verteilte Zahlenfolge:* Erzeugen Sie eine Zahlenfolge mit logarithmischer Verteilung und 10 Werten zwischen 0,1 und 5,5. *Hinweis:* Den Logarithmus zur Basis 10 berechnet man mit dem Befehl: `log10`.

Variablen

Wie in jeder Programmiersprache können auch in MATLAB Variablen definiert werden. Die Zuweisung erfolgt mit dem Gleichheitszeichen =. Jede Zeichenfolge, die mit einem Buchstaben beginnt und keine Sonderzeichen enthält, darf als Variablenname verwendet werden. Groß- und Kleinschreibung wird unterschieden. MATLAB benötigt keine vorangehenden Deklarationen, d. h. eine Variable wird automatisch neu deklariert, wenn sie auf der linken Seite einer Zuweisung verwendet wird. Auf der rechten Seite dürfen nur bereits bekannte Variablen oder Konstanten verwendet werden. Rekursive Zuweisungen wie beispielsweise `a = a + 1` sind möglich.

Wie bereits erwähnt, basieren die Variablen in MATLAB auf Matrizen. Diese können sowohl Zahlen als auch Buchstaben enthalten. In diesem Abschnitt beschränken wir uns auf die wesentlichen numerischen Elemente (Matrizen, Vektoren und Skalare), auf Zeichen und auf so genannte Zeichenketten.

Analog zum Workspace Browser wird mit den Befehlen `who` (Namen der verwendeten Variablen) und `whos` (Namen und Größen der verwendeten Variablen) der Inhalt des Workspace im Befehlsfenster angezeigt.

```
>> A = [1,2,3;4,5,6;7,8,0] % Zuweisung einer Matrix der Variable A
>> b = [1;2;3]             % Zuweisung eines Vektors der Variable b
>> A*b                     % Matrixmultpilikation
>> d = 2*A;                % Multiplikation mit einem Skalar
>> who                     % Vorhandene Variablen?
>> whos                    % und ihre Eigenschaften?
```

`ans`	Enthält das letzte Ergebnis eines Befehls, falls keine Zuweisung zu einer anderen Variable erfolgt ist.
`pi`	Enthält die Zahl π.
`eps`	Enthählt die Zahl $=2^{-52}$.
`realmin, realmax`	Kleinstee und größter reeller Wert, der in MATLAB verfügbar sind.
`Inf`	(infinit: unendlich) steht für 1/(+0). Beachten Sie den Unterschied zu 1/(-0).
`NaN`	(Not a Number) steht für 0/0.
`i, j`	Enthalten beide die imaginäre Einheit $\sqrt{-1}$ zur Eingabe von komplexen Zahlen.

Tabelle 3.1: Einige reservierte Systemvariablen

Außerdem gibt es noch einige reservierte, vorbesetzte Systemvariablen (siehe Tabelle 3.2), die nicht verändert werden sollten. Falls aus Versehen eine Systemvariable *vname* mit einem anderen Wert überschrieben worden ist, so erhält man den ursprünglichen Wert mit dem Befehl `clear` *vname* wieder, der die entsprechende Variable aus dem Workspace löscht.

```
>> pi = 2.3
>> clear pi
>> pi
>> komplex = 3 + 2 * i
```

Möchte man auf einzelne Werte aus einer Matrix zugreifen, so werden die entsprechenden Zeilen- und Spaltennummern in runde Klammern gesetzt. Die Zeilen- und Spaltennummern können auch wieder Vektoren sein, um gleichzeitig auf mehrere Zeilen oder Spalten zugreifen zu können.

```
>> A(1,3)
>> A(1,:)      % ':' repräsentiert ganze Zeile
>> A(:,2)      % ':' repräsentiert ganze Spalte
>> A([1 3],1)  % Elementen der 1. Spalte und 1. und 3. Zeilen
```

Die sogenannten **Logischen Matrizen** (LM) helfen, Werte, die bestimmten Bedingungen genügen, aus einer Matrix A zu extrahieren. Dazu muss die logische Matrix mit der gleichen Dimension wie A und der entsprechenden Bedingung, z. B. Werte zwischen 2 und 5, aufgebaut werden. Anschließend können die Werte extrahiert werden. Das folgende Beispiel verdeutlicht das Vorgehen:

```
>> X=(A>=2 & A<=5)    % LM mit 1, wo die Bedingung erfüllt wird
>> A(X)               % Extraktion der entsprechenden Elemente (Vektor)
>> A.*X               % Extraktion der entsprechenden Elemente (Matrix)
```

Mit Hilfe einer leeren Matrix [ ] können wieder Zeilen oder Spalten einer Matrix gelöscht werden:

```
>> A(:,2) =[ ]        % Die zweite Spalte von A wird gelöscht.
```

Aufgaben und Übungen

3.14 ☞ *Vektoren, Matrizen und der Workspace:* Erstellen Sie zwei Variablen **Y**= $\begin{bmatrix} 1 & 5 & 7 \\ 2 & 5 & \pi \end{bmatrix}$, **t**= $\begin{bmatrix} 2 & 4 & 6 & \dots & 14 \end{bmatrix}$ und verändern Sie ihren Inhalt durch einen Doppelklick auf die Variable im Workspace-Fenster. Überprüfen Sie die Änderung, indem Sie die Variablennamen im Befehlsfenster eingeben.

- Erstellen Sie einen Spaltenvektor **t2** mit logarithmischer Teilung und Werten von 10^{-3} bis 10^{0}. *Hinweis:* Um einen Zeilenvektor in einen Spaltenvektor umzuwandeln, nutzen Sie den Traspositionsoperator `'` (z.B. `vektor'`).
- Extrahieren Sie die erste Zeile **z1** und die dritte Spalte **c3** von **Y**.
- Extrahieren Sie die Matrix **Yklein** mit der 1. und 3. Spalte von **Y**.
- Extrahieren Sie den Vektor **zwischen** mit Werten größer als 3 aus der Matrix **Y**.
- Löschen Sie die zweite Zeile von **Y**.

Laden, Speichern und Löschen

Es gibt in MATLAB auch die Möglichkeit, die Variablen im Workspace zu verwalten und vor allem für spätere Arbeiten zur Verfügung zu stellen. Einige der dafür benötigten Befehle sehen Sie in Tabelle 3.2).

`clear`	Löscht alle Variablen im Workspace.
`clear` *variable*	Löscht *variable* aus dem Workspace.
`save`	Speichert alle Variablen in Datei `matlab.mat`.
`save` *fname*	Speichert alle Variablen in Datei `fname.mat`.
`save` *fname x y z*	Speichert Variablen x, y und z in Datei `fname.mat`.
`load`	Lädt alle Variablen aus Datei `matlab.mat` in den Workspace.
`load` *fname*	Lädt alle Variablen aus Datei `fname.mat` in den Workspace.
`load` *fname x y z*	Lädt Variablen x, y und z in den Workspace.

Tabelle 3.2: Einige reservierte Systemvariablen

Von diesen Befehlen sind hier die wichtigsten Optionen vorgestellt, sie besitzen aber noch weitere. Im Allgemeinen sei mit dieser Einführung immer wieder auf die MATLAB-Hilfe verwiesen, um alle Möglichkeiten kennenzulernen und auszunutzen.

Aufgaben und Übungen

3.15 ☞ *Laden, löschen und speichern:* Speichern Sie alle Variablen aus der Aufgabe 3.14 in eine Datei und löschen Sie den Workspace mit `clear all`. Überprüfen Sie den Inhalt des Workspace (z. B. `who`). Laden Sie Ihre Variablen aus der Datei erneut in den Workspace durch Doppelklicken oder mit dem Befehl `load`.

3.5 Operatoren und eingebaute Funktionen

Operatoren

Die rechte Seite einer Zuweisung darf ein beliebig komplizierter algebraischer Ausdruck sein. Konstanten und bekannte Variablen dürfen in sinnvoller Weise mit Operatoren verknüpft werden. Bei den arithmetischen Operatoren sind neben den üblichen (`+`,`-`,`*`,`/`) insbesondere das Potenzieren (`^`) und Transponieren (`'`) von Bedeutung. Daneben gibt es auch die logischen Operatoren

~ (nicht), & (und), | (oder), xor() (exklusiv oder) und die relationalen Operatoren == (gleich), ~= (ungleich), <, ≤(in MATLAB <=), >, ≥(in MATLAB >=). Es gelten die üblichen Regeln für Punkt- vor Strichrechnungen. Andere Gruppierungen können mit runden Klammern ( ) erreicht werden. Eine Übersicht über die Operatoren liefert der Befehl:

```
>> help ops
```

Achtung: Die Operatoren beziehen sich auf Matrixberechnungen. Möchte man den Operator auf die einzelnen Elemente einer Matrix (so genannte **elementweise Operationen**) anwenden, so muss dem Operator ein Punkt vorangestellt werden!

```
>> A = [1, 2, 3; 4, 5, 6; 7, 8, 0] % Matrix
>> b = [5,6,7]                     % Zeilenvektor
>> b=b'                            % Spaltenvektor
>> b+b                             % Vektorsumme
>> b'*b                            % Skalarprodukt
>> b*b                             % Fehler: inkompatible Dimension
>> b.*b                            % Elementweise Multiplikation
>> A*A                             % A mal A und
>> A^2                             % A hoch 2 sind identisch
>> A.^2                            % aber nicht hiermit!!!
```

Eingebaute Funktionen

Wie schon implizit in den vorigen Abschnitten gezeigt, enthält MATLAB sehr viele Funktionen aus verschiedensten Gebieten. Ein Funktionsaufruf besteht aus einem Namen und Übergabeparametern. Das Ergebnis kann einer Variablen oder einem Vektor von Variablen zugewiesen werden. Eine Übersicht über mathematische Standardfunktionen liefert der Befehl `help elfun`.

Zwei sehr nützliche Funktionen sind `size` und `length`. `size` gibt die Anzahl der Zeilen und Spalten einer Matrix an und ermöglicht somit, Zeilen- von Spaltenvektoren zu unterscheiden. `length` gibt die Länge eines Vektors an und im Falle einer Matrix den größeren Wert der Spalten- und Zeilenanzahl.

```
>> A=[1, 2, 3; 4, 5, 6]
>> size(A)
>> length(A)
>> size(b)
>> length(b)
```

Eine Übersicht über Standardmatrizen und Matrizenmanipulationen liefert der Befehl `help elmat` sowie über Funktionen aus dem Bereich der linearen Algebra `help matfun`. Die Funktionen `ones` und `zeros` erstellen Matrizen mit nur Einsen bzw. Nullen als Einträgen und die Funktion `eye` erstellt die Einheitsmatrix.

Aufgaben und Übungen

3.16 ☞ *Erzeugen von Standardmatrizen:* Erzeugen Sie eine Matrix E, die vollständig mit 1 belegt ist und die Dimension 2x3 hat. Erstellen Sie eine 4x4-Einheitsmatrix I. Geben Sie die Anzahl der Zeilen und Spalten von E und I an.

3.17 ☞ *Determinante und Eigenwerte:* Berechnen Sie die Determinante (*determinant*) der Matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 1 \\ 7 & 8 & 9 \end{bmatrix}$ und ermitteln Sie ihre Eigenwerte (*eigenvalues*). Finden Sie zunächst die dafür benötigten Befehle. Ist diese Matrix invertierbar? Warum? Berechnen Sie die Inverse der Matrix A^{-1} und ihre Determinante.

3.18 ☞ *Lösungen eines linearen Gleichungssystems:* Gegeben ist die Gleichung: $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 1 \\ 7 & 8 & 9 \end{bmatrix} \cdot x = \begin{bmatrix} 6 \\ -7 \\ 0 \end{bmatrix}$. Bestimmen Sie die Lösung `x` des Gleichungssystems einmal mit Hilfe der Inversen der Matrix und einmal mit Hilfe des Operators `\`, der zum Lösen von linearen Gleichungssystemen gedacht ist. Man löst eine Gleichung der Gestalt $A * x = b$ mit dem Befehl `A\b` (Gauß-Eliminationsverfahren).

3.6 Logische Operatoren und bedingte Anweisungen

Wie in jeder anderen Programmiersprache auch, besitzt MATLAB die Möglichkeit, bedingte Anweisungen zu realisieren. Diese sind

- **if** ... **elseif** ... **else** ... **end**

  Die `if`-Anweisung gestattet die Ausführung einer oder mehrerer Anweisungen in Abhängigkeit von einer Bedingung. Bei der Bedingung handelt es sich um einen logischen Ausdruck, der nach dem Schlüsselwort `if` in Klammern anzugeben ist. Die Syntax der `if`-Anweisung lautet:

```
if (Bedingung)
    Anweisungen
end
```

```
if (Bedingung 1)
    Anweisungen 1
else
    Anweisungen 2
end
```

```
if (Bedingung 1)
    Anweisungen 1
elseif (Bedingung 2)
    Anweisungen 2
else
    Anweisungen 3
end
```

  Für die Argumente der `if`-Anweisungen werden in der Regel logische Ausdrücke verwendet .

```
>> a <= b       % ist a kleiner gleich b?
>> a == b       % ist a gleich b?
>> a ~= b       % ist a ungleich b?
>> isempty(a)   % ist die Variable a leer?
>> ischar(a)    % ist a ein Zeichen?
>> isnumeric(a) % ist a eine Zahl?
>> exist(a)     % ist die Variable, Datei, etc. vorhanden?
```

- **switch** ... **case** ... **otherwise** ... **end**

  Die `switch`-Anweisung eignet sich besonders dann, wenn mehrere Anweisungsteile alternativ ausgeführt werden sollen. Es erfolgt keine Bedingungsprüfung, sondern ein direkter Wertevergleich, was den Anwendungsbereich der `switch`-Anweisung etwas einschränkt. Die Syntax lautet:

```
switch(Ausdruck)
    case Konstante1:
        Anweisung(en)
    case Konstante2:
        Anweisung(en)
    otherwise
        Anweisung(en)
end
```

- **try . . . catch . . . end**

  Die `try-catch`-Anweisung wird dann verwendet, wenn nicht sichergestellt werden kann, dass ein bestimmter Teil des Programms für alle Eingaben richtig abläuft bzw. wenn sicher ist, dass er nicht für alle Eingaben richtig laufen wird. Um nicht alle möglichen Fälle durch `if`-Anweisungen abzudecken (was auch nicht immer möglich sein wird), gibt es die `try-catch`-Anweisung. Zunächst wird der Teil nach `try` ausgeführt und falls ein Fehler auftritt, wird der Anweisungsteil nach `catch` ausgeführt. Falls dann im `catch`-Teil ein Fehler auftritt, wird das Programm abgebrochen. Die Syntax ist folgendermaßen:

```
try
    Anweisungen, die im Normalfall ausgeführt werden
catch
    Anweisungen, die im Fall eines Fehlers im try-Teil ausgeführt werden
end
```

Aufgaben und Übungen

3.19 ☞*isEven*: Schreiben Sie eine Matlab-Funktion `isEven.m`, welche eine ganze Zahl `n` bekommt und dann ausgibt, ob die Zahl gerade ist (Rückgabe von `Ja`) oder nicht (Rückgabe von `Nein`). Nutzen sie dafür eine `if` . . . `else`-Anweisung. *Hinweis*: Um zu überprüfen, ob eine Zahl durch zwei teilbar ist, können Sie die Funktion `mod` benutzen.

3.20 ☞*Lebensphase*: Schreiben Sie ein Programm, welches folgenden Text ausgibt:

```
In welcher Lebensphase befinden Sie sich jetzt!

Drücken Sie 1, wenn Sie zw. 0 und 12 Jahre alt sind.
Drücken Sie 2, wenn Sie zw. 13 und 21 Jahre alt sind.
Drücken Sie 3, wenn Sie zw. 22 und 59 Jahre alt sind.
Drücken Sie 4, wenn Sie älter als 60 Jahre sind.
```

Nach der Nutzereingabe (`input`), soll mit Hilfe von `switch` . . . `case` eine entsprechende Meldung angezeigt werden.

```
Sie sind noch ein Kind!
Sie sind ein Teenager!
Sie sind ein Erwachsener!
Sie sind ein Senior!
```

Betrachten Sie auch den Fall einer falschen Eingabe und geben Sie eine passende Fehlermeldung aus.

3.21 ☞*Laden*: Schreiben Sie mit Hilfe von `try ... catch ... end` eine Funktion `Laden(name)`, die eine `.mat`-Datei laden soll und die Fehlermeldung *Fehler: Datei nicht vorhanden!* ausgibt, falls diese Datei nicht vorhanden ist. Versuchen Sie jetzt eine nicht existierende Datei zu laden: `Laden('mich_gibt_es_nicht.mat')`. Zum Vergleich versuchen Sie die nicht existierende Datei `mich_gibt_es_nicht.mat` direkt mit dem Befehl `load` zu laden.

3.7 Schleifen

Eine Schleife ist wie die bedingten Anweisungen eine Kontrollstruktur in Programmiersprachen. Sie wiederholt einen Code-Block, den so genannten Schleifenrumpf, so lange, wie eine Laufbedingung gültig ist oder bis eine Abbruchbedingung eintritt. Schleifen, deren Laufbedingung immer erfüllt ist oder die ihre Abbruchbedingung niemals erreichen, und Schleifen, die keine Abbruchbedingungen haben, sind **Endlosschleifen**.

Die Schleifen werden in MATLAB mit folgenden Befehlen aufgebaut:

- **for ... end**

  Die `for`-Schleife eignet sich besonders dann, wenn die Gesamtzahl der Wiederholungen von vornherein feststeht. Dementsprechend ist sie als typische Zählschleife vorgesehen.Ihre Syntax lautet:

  ```
  for(Initialisierung:Schrittweite:Endwert)
      Anweisung(en)
  end
  ```

  Wird keine explizite Schrittweite vorgegeben, so verwendet MATLAB die Schrittweite 1.

- **while ... end**

  Bei einer `while`-Schleife wird der Schleifenrumpf solange wiederholt, bis die logische Operation einen falschen Rückgabewert liefert. Wird der Inhalt der logischen Operation nicht im wiederholten Teil des Programmcodes verändert, ist diese Kontrollstruktur meist nicht die richtige, weil diese Schleife sonst kein einziges Mal durchlaufen wird oder unendlich lang läuft (für Endlosschleifen wird oft `while(1)` benutzt).

  ```
  while(Bedingung)
      Anweisung(en)
  end
  ```

Aufgaben und Übungen

3.22 ☞ *Vektoren vs. Schleifen*: Schreiben Sie ein Skript, das die Werte der untenstehenden Funktion $y(t)$ in Abhängigkeit von `t=-pi:pi/1e4:4*pi` berechnet. Die Werte sollen zunächst nur mit `if` und `for` berechnet und in einem Vektor gespeichert werden.

$$y(t) = \begin{cases} 0 & \forall \quad t < 0 \\ t & \forall \quad 0 \leq t < 1 \\ 0.1 \cdot sin(2\pi \cdot t) + 1 & \forall \quad t \geq 1 \end{cases}$$

In demselben Skript wiederholen Sie jetzt die Aufgabe anhand von Vektoroperationen und ermitteln die benötigte Rechenzeit mit den Befehlen `tic` und `toc`. Der Befehl `tic` startet eine Uhr, der Befehl `toc` stoppt die Uhr und gibt die verstrichene Zeit im Command Window aus. Ermitteln Sie die Rechenzeit auch für den Fall ohne Vektoroperationen.

Welche Methode ist schneller? Warum?

3.23 ☞ *Fakultät einer ganzen Zahl s als Skript*: Schreiben Sie ein Skript, mit dem die Fakultät einer Zahl s berechnet wird. Hierbei sollte zunächst die Zahl s deklariert werden und dann mit einer `for`-Schleife die Fakultät berechnet werden. *Zur Erinnerung:* Die Fakultät einer ganzen Zahl n berechnet man folgendermaßen: $n! = n \cdot (n-1)! \Rightarrow n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$

3.24 ☞ *Erweiterung des Faktultät-Skripts*: Erweitern Sie das Skript zur Berechnung der Fakultät einer ganzen Zahl s so, dass ein maximales ganzzahliges n ermittelt wird, für das gilt $n! < 1000$.

3.25 ☞ *Interaktives Programm:* Schreiben Sie ein Programm, das die Reaktionszeit einer Person beim Tastendrücken misst.

Mit Hilfe der Befehle `rand` und `round` soll dem Nutzer eine ganze Zahl zwischen 0 und 10 gezeigt werden, welche er sofort auf der Tastatur eingeben soll, während die vergangene Zeit gemessen wird `[tic, toc]`. Das Verfahren soll zehnmal wiederholt werden.

Der Mittelwert der Reaktionszeit (zw. Zeigen und Eingeben) wird als Ergebnis in einem separaten Fenster `[msgbox]` dargestellt. Falls die Person eine falsche Eingabe macht, muss das Programm mit einer Nachricht abgebrochen werden.

Erweitern Sie das Programm so, dass die Reaktionszeiten pro Versuch geplottet werden und dass folgende von der Reaktionszeit (Mittelwert) abhängige Nachrichten angezeigt werden:

$T \leq 0.5\ Sek$	Du bist sehr schnell !!
$T \leq 1\ Sek$	Du bist schnell !!
$T \leq 2\ Sek$	Nicht sehr beeindruckend !!
$T > 2\ Sek$	Eine Schildkröte wäre schneller als du !!

3.8 Graphische Darstellungen

MATLAB besitzt auch eine Reihe von graphischen Möglichkeiten. Die wichtigsten Graphikfunktionen sind:

- 2D-Kurven: `plot, fplot, ezplot, subplot, stem, stairs`
- 3D-Kurven: `plot3, stem3, surf, mesh, contour`

Der Befehl 'plot'

Möchte man einen Vektor **y** graphisch darstellen, so lautet der Befehl `plot(y)`. In dem Fall werden die Elemente von **y** (y_1, y_2,y_3,...) über ihre Indizes (1,2,3...) 2-dimensional aufgetragen. Möchte man eine andere x-Achse, z. B die Zeit t, so muss dieser Vektor **t** dem Vektor **y** vorangestellt werden,

`plot(t,y)`. Der entsprechende Befehl für die dreidimensionale Darstellung, bei dem immer alle drei Vektoren angegeben werden müssen, lautet `plot3(t,x,y)`.

```
>> t = linspace(-pi, pi, 30);
>> y1= sin(2 *t) ;
>> y2= cos(5 *t) ;
>> plot(y1)
>> plot(t, y1)                    % Zum Vergleich
>> plot(y1, t)                    % Zum Vergleich
>> plot3(t, y1, y2)
```

Die Zeichnungen werden in sogenannten **figure**s dargestellt. Dies sind eigene Fenster, die neben der Kurve noch die Achsen, verschiedene Menübefehle zum Ändern des Layouts usw. enthalten. Ist bereits ein figure-Fenster offen, so wird die Kurve in dieses Fenster geplottet und die alte Kurve gelöscht. Mit `hold` wird/werden die alte(n) Kurve(n) beibehalten und die neue dazu geplottet. Ist noch kein figure-Fenster offen oder der Befehl `figure` dem plot-Befehl vorangestellt, wird ein neues Fenster erzeugt.

Die meisten Layout-Einstellungen sind nicht nur über das figure-Menü, sondern auch über das Befehlsfenster zu ändern. Nützliche Befehle sind:

```
>> grid on/off % Schaltet ein Hintergrundraster (grid) ein/aus
>> box on/off  % Schaltet eine umrahmende Box ein/aus
>> axis on/off % Schaltet alle aktuellen Achsendarstellungen ein/aus
```

Wird on/off bei den ersten beiden Befehlen weggelassen, so wird zwischen den beiden Zuständen (an/aus) hin- und hergeschaltet.

Der Plotbefehl lässt ein zusätzliches Argument, eine Zeichenkette, zu, z. B. `plot(t,y1,'g')`. In dieser Zeichenkette können Angaben zur Farbe, Liniendarstellung und Punktedarstellung gemacht werden. Für eine genaue Beschreibung sowie weiterer Möglichkeiten rufen Sie bitte die Hilfe mit `help plot`, `help graph2d` und `help graph3d` auf.

Ein **figure-Fenster** kann wie eine Matrix in mehrere „Unter-Bilder"mit dem Befehl `subplot` aufgeteilt werden:

```
>> figure                      % öffnet ein neues figure-Fenster
>> subplot(2,1,1); plot(t,y1)
                               % subplot teilt das Fenster in 2
                               % Zeilen und 1 Spalte auf und
                               % wählt das erste Element (es wird
                               % von links nach rechts und von
                               % oben nach unten  gezählt) zur
                               % Darstellung der nächsten
                               % geplotteten Kurve aus
>> subplot(2,1,2); plot(y1,t)
                               % subplot benutzt die gleiche Auf-
                               % teilung, wählt aber jetzt
                               % das zweite Fenster aus
```

Zur einfachen Beschriftung existieren folgende Funktionen, welche auf den gerade aktuellen Graphen angewendet werden:

```
>> title('Schöne Kurve')   % überschreibt den Graph mit einem Titel
>> xlabel('Zeit /s')       % beschriftet die x-Achse
>> ylabel('z /m')          % beschriftet die y-Achse
```

Enthält ein Graph mehrere Kurven, kann mit folgendem Befehl eine Legende erzeugt werden.

```
>> legend('Kurve1','Kurve2',...)
```

Andere Graphen

Die Funktionen `stem` bzw. `stem3` und `stairs` sind für die Darstellung diskreter Signale geeignet. Dabei stellt stem jeden Punkt durch einen senkrechten Balken mit einem Kreis an der Spitze dar. Stairs ist die Treppenfunktion und verbindet die einzelnen Punkte wie eine Treppe, indem der Wert des vorigen Punkts bis zum nächsten beibehalten wird.

`fplot` berechnet in einem angegebenen Intervall die Werte und den Kurvenverlauf einer Funktion, ohne vorher die genaue Anzahl an Punkten anzugeben.

```
>> stem(y1)
>> stairs(t,y1)
>> fplot('3*sin(r).* exp(-r/(pi/4))', [0, 2* pi]);
```

Darstellung von Oberflächen

Zur Darstellung von Oberflächen werden die Befehle `mesh` und `surface` benutzt:

- `mesh` plottet ein buntes Netz, bei dem die Knoten die Punkte der angegebenen Funktion sind.
- `surf` stellt auch die Maschen des Netzes, also die gesamte Oberfläche, bunt dar.

Die Farbpalette wird über `colormap` festgelegt und kann mit Hilfe von `colorbar` angesehen werden.

Anhand eines Beispiels wird verdeutlicht, wie die benötigten Vektoren erzeugt werden. Wir möchten die Oberfläche des Graphen der Funktion $z = -2x + 3y$ darstellen, wobei x und y die folgenden Werte annehmen sollen:

```
>> x = [-1, 0, 1, 2]
>> y = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

Da x 4 und y 6 Werte enthält, muss z $6 \times 4 = 24$ Werte annehmen. Darüber hinaus müssen, da jeder Punkt durch ein Tripel (x_i, y_i, z_i) beschrieben wird, die 6×4-Matrizen X und Y erzeugt werden, die in jeder Zeile bzw. Spalte die Werte von x und y enthalten. Dies geschieht am einfachsten mit dem Befehl `meshgrid`. Anschließend kann `z` berechnet und alles gezeichnet werden:

```
>> x = [-1 : 2]; y = [0 : 0.1 : 0.5] % Definition zweier Vektoren
>> [X, Y] = meshgrid (x,y);          % Erzeugen der Matrizen X und Y
>> Z = -2.*X + 3.*Y;                 % Berechnen der Matrix Z
>> mesh ( X, Y, Z)                   % oder auch surf ( X, Y, Z)
```

Abschließend siehe auch die Befehle: `compass, contour, contour3, surfc, waterfall, pcolor` und `view`.

Aufgaben und Übungen

3.26 ☞ *Erzeugen und Änderung von Abbildungen:*

Erzeugen Sie die Matrix **GraphA**= $\begin{bmatrix} 1 & 2 & 3 & \dots & 6 \\ 1 & 4 & 9 & \dots & 36 \\ 1 & 8 & 27 & \dots & 216 \end{bmatrix}^{\mathbf{T}}$ und plotten Sie **GraphA**. Fügen Sie dem Graph eine Legende mit den Texten „linear“, „quadratisch“ und „kubisch“ hinzu.

Schalten Sie das Hintergrundraster ein. Stellen sie über das Menü *Edit:Axes Properties* die Achsenskalierung der y-Achse auf logarithmisch um. Geben Sie der Kurve den Titel „GraphA“.

3.27 ☞ *Plotten einer Funktion in 3D:* Plotten Sie die Funktion $z = y * sin(x)$ für x zwischen -10 und 10 (Inkrement 1) und y zwischen 0 und 30 (Inkrement 3) und stellen Sie danach ihre Oberfläche als Netz dar.

3.28 ☞ *Schiefer Wurf ohne Reibung* : Berechnen Sie die Flugbahn einer punktförmigen Masse M in der (x,z)-Ebene, die im Punkt $(0,h)$ mit der Geschwindigkeit (v_{x0},v_{z0}) losgeworfen wird:

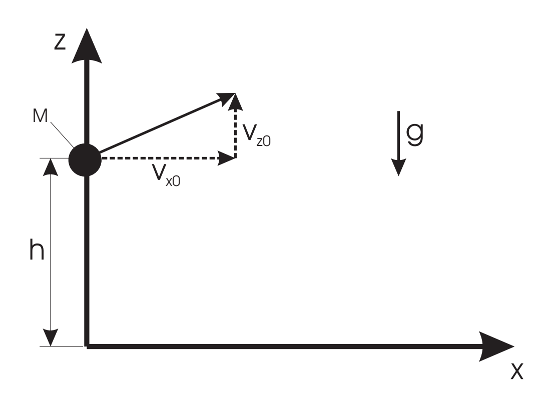


Abbildung 3.5: Schiefer Wurf

Vernachlässigen Sie zunächst alle Reibungs- und Kontakteffekte.

Es gilt:

Parameter	Wert
M	2 kg
v_{x0}, v_{z0}	10 $\frac{\text{m}}{\text{s}}$
h	5 m
g	9.81 $\frac{\text{m}}{\text{s}^2}$

Das System kann durch folgende kontinuierliche Differentialgleichungen beschrieben werden:

$$\begin{aligned} \dot{x} &= v_{x0} \\ \ddot{z} &= -g \end{aligned}$$

- Schreiben Sie ein Matlab-Skript, das die Geschwindigkeiten und Positionen der Masse für Zeitschritte $\Delta t = 0.1$ sec bis $t_{\text{end}} = 5$ sec berechnet und in einem Vektor ablegt. Nutzen Sie dazu eine `for`-Schleife, um die Geschwindigkeit und die Position in jedem Zeitpunkt aus der Geschwindigkeit und der Position zum vorhergehenden Zeitpunkt zu berechnen.

- Speichern Sie das Skript unter dem Namen `SchieferWurf.m`.
- Zeichnen Sie die Flugbahn mithilfe des Befehls `plot`.

3.29 ☞ *Schiefer Wurf mit Boden*: Ergänzen Sie Ihr Skript aus der Aufgabe 3.28 um einen „Boden“ $z = 0$. Modellieren Sie den Kontakt als idealen Stoß, d.h. die vertikale Geschwindigkeit der Masse wird invertiert $v_{z,i} = -v_{z,i-1}$.

3.30 ☞ *Schiefer Wurf mit Reibung*: Ergänzen Sie das Modell um Luftreibung.

- Der Betrag der Luftreibungskraft ergibt sich zu

$$|\vec{F}_L| = c_L|\vec{v}|^2 = c_L \left| \begin{pmatrix} \dot{x} \\ \dot{z} \end{pmatrix} \right|^2,$$

  wobei $c_L = 0.01\ \frac{\mathrm{Ns}^2}{\mathrm{m}^2}$. Ihre Richtung weist entgegen dem Geschwindigkeitsvektor $\vec{v} = \begin{pmatrix} \dot{x} \\ \dot{z} \end{pmatrix}$.

- Die Kraft F_L sorgt für eine zusätzliche Beschleunigung der Masse

$$\begin{pmatrix} \ddot{x} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ -g \end{pmatrix} - \frac{1}{M} F_L$$

3.31 ☞ *Plotten symbolischer Funktionen :* Plotten Sie mit dem Befehl `fplot` die Funktion *z=sin(x)/x* für den Bereich $-6\pi < x < 6\pi$. In ähnlicher Form plotten Sie mit Hilfe des Befehls `ezplot` die Funktion $4x^2 + 3xy + y^2 = 4$.

3.9 Funktionen

Funktionen besitzen einen eigenen Workspace mit lokalen Variablen und haben Ein- und Ausgangsargumente. Neben selbstgeschriebenen können auch bereits mit MATLAB/SIMULINK mitgelieferte Funktionen, die in MATLAB (und nicht z. B. C) geschrieben sind, editiert werden, z. B. *edit log10*. Folgende Syntax muss bei einer Funktion beachtet werden:

```
function [Ausgangsargumente] = funktionsname(Eingangsargumente)
% Alles, was hier angegeben wird, erscheint, falls man die Hilfe zu
% dieser Funktion mit 'help funktionsname' aufruft.

Liste der funktionseigenen Befehle
```

Der Unterschied zwischen einer Funktion und einem Skript-File besteht darin, dass die im Funktions-File definierten Variablen lokal sind. Nach Auswertung der Anweisungen im Funktions-File sind sie also im Workspace unbekannt. Umgekehrt kann das Funktions-File auch auf keine Variablenwerte aus dem Workspace zugreifen.

Der angegebene `'funktionsname'` muss dabei mit dem Namen der Datei `'funktionsname.m'` übereinstimmen. Die erste Zeile darf nicht mit einem Semikolon abgeschlossen werden. Die zweite und folgenden anschließenden Zeilen, die mit einem % beginnen, werden bei Aufruf der Hilfe zu dieser Funktion ausgegeben.

Zur Verdeutlichung folgt ein einfaches Beispiel. Schreiben Sie folgende Anweisungen in eine Datei mit dem Namen `'lingls.m'`:

```
function [x, detA] = lingls(A,b)
% LINGLS löst das lineare Gleichungssystem A*x=b und gibt
% den Loesungsvektor x sowie die Determinante der Matrix A zurück.

detA=det(A); x=A\b;
```

Führen Sie anschließend die Befehle aus:

```
>> help lingls
>> matA = [1, 2, 3; 4, 5, 6; 7, 8, 0];
>> vek = [2; 5; 9];
>> lingls(matA, vek);
>> [antw1, antw2] = lingls(matA, vek);
```

Möchte man eine Funktion schreiben, die mit einer unterschiedlichen Anzahl an Argumenten aufgerufen werden kann, sind die MATLAB-Funktionen `nargin` und `nargout` hilfreich. Sie liefern die Anzahl der Ein- und Ausgangsargumente. Ihre Verwendung ist in der Funktion `bode` gut zu erkennen: `edit bode`.

Möchte man die in einer Funktion verwendeten Variablen, z. B. t und x, auch anderen Funktionen zur Verfügung stellen oder im Workspace sichtbar machen, müssen sie an jeder Stelle (d. h. in den entsprechenden anderen Funktionen, aber auch im Workspace) als `global` deklariert werden:

```
>> global t, x
```

Dies stellt aber programmiertechnisch eine unschöne Lösung dar. Die bessere Programmierung ist die Aufnahme als Ausgangsargument.

Aufgaben und Übungen

3.32 ☞ *Funktion mit zwei Vektorargumenten:* Schreiben Sie eine Funktion, die

- als Eingangsargumente zwei Vektoren erhält
- überprüft, ob es sich wirklich um Vektoren und nicht um Matrizen handelt
- im Falle der Eingabe einer Matrix eine Fehlermeldung in dem Befehlsfenster ausgibt und den Wert Null als Ausgangs-Argument zurückgibt
- sonst den größten Wert beider Vektoren berechnet und diesen als Ausgangs-Argument zurückgibt
- Verfassen Sie dazu einen entsprechenden Hilfe-Text.

3.33 ☞ *Abschnittsweise definierte Funktion:* Schreiben Sie eine Funktion, die den Wert der abschnittsweise definierten Funktion in Abhängigkeit von t berechnet:

$$y(t) = \begin{cases} 0 & \forall \quad t < 0 \\ t^2 \cdot (3 - 2 \cdot t) & \forall \quad 0 \leq t \leq 1 \\ 1 & \forall \quad t > 1 \end{cases}$$

Stellen Sie die Funktion im Bereich von -1 bis 3 graphisch dar. *Hinweis:* t kann ein Vektor sein.

3.34 ☞ *Fakultät einer ganzen Zahl s als Funktion*: Die Fakultät der Zahl s soll im Gegensatz zur Aufgabe 3.26 mit Hilfe einer Funktion berechnet werden. Schreiben Sie eine Funktion `fakultaet(s)`, die jeweils die Fakultät der Zahl s als Ergebnis zurückgibt. Dabei sollten Sie beachten, was für die Fakultät für die verschiedenen Eingaben herauskommt. *Tipp:* Es kann auch eine Rekursion verwendet werden. Das heißt, die Funktion muss sich selbst wieder aufrufen.

3.35 ☞ *Rechnen mit der Fakultät*: Es gibt nun die Möglichkeit, mit der Funktion `fakultaet(s)` die Fakultät zu berechnen. Nun wollen wir herausfinden, welche ganze Zahl diejenige ist, deren Fakultät gerade noch kleiner als 1000 ist. Die Aufgabe lautet also: Finden Sie das größte n, so dass gilt $n! < 1000$.

Hierfür gibt es jetzt zwei Varianten: Zum einen können Sie diese Aufgabe lösen, indem Sie für jede Zahl mit Hilfe der Funktion `fakultaet(s)` die Fakultät berechnen und überprüfen, ob die Fakultät noch kleiner als 1000 ist. Sie können das Resultat der Aufgabe aber mit weniger Rechenzeit erhalten, indem Sie die Fakultät mit Hilfe einer Schleife (ohne Funktion) berechnen und in jedem Schritt der Fakultätsberechnung überprüfen, ob die Fakultät noch kleiner als 1000 ist. Vergleichen Sie die Zeit, die das Programm benötigt, um das Resultat der Aufgabe zu bekommen, mit der, die man bei Berechnung mit der Funktion benötigt. Hierfür können Sie die Befehle `tic` und `toc` verwenden.

3.36 ☞ *isPrime*: Schreiben Sie eine Funktion `isPrime(zahl)`, die eine ganze Zahl übergeben bekommt und überprüft, ob die Zahl eine Primzahl ist oder nicht. *Zur Erinnerung:* Eine Primzahl ist eine natürliche Zahl, die nur zwei Teiler hat, nämlich 1 und sich selbst. Die Zahl 1 ist jedoch keine Primzahl.

3.37 ☞ *100 Primzahlen in einem Vektor*: Erzeugen Sie nun einen Vektor, in dem nach Ablauf des Programms die ersten 100 Primzahlen stehen. Um zu überprüfen, ob eine Zahl eine Primzahl ist, können Sie die Funktion `isPrime(zahl)` aus der vorangegangenen Aufgabe benutzen. Anschließend speichern sie den Vektor in der Datei `primzahlen.mat` ab.

3.38 ☞ *Fibonacci*: Programmieren Sie eine Funktion, die die n-te Fibonacci-Zahl f_n berechnet. Die Fibonacci-Zahl ist folgendermaßen definiert:
$f_0 = f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$
Somit ergeben sich die ersten Fibonacci-Zahlen wie folgt:
$f_0 = f_1 = 1$
$f_2 = 1 + 1 = 2$
$f_3 = 2 + 1 = 3$
$f_4 = 3 + 2 = 5$

3.10 Strukturen in Matlab

Strukturen dienen zur Erstellung eigener komplexer Datenstrukturen oder eigener Datentypen. So kann zum Beispiel eine Raumkoordinate, bestehend aus einer X-, einer Y- und einer Z-Position, durch eine Datenstruktur Koordinate abgebildet werden. Es können außerdem Daten verschiedener Typen in Strukturen zusammengefasst werden, so könnte man eine Struktur mit Name, Matrikelnummer und Semester erzeugen, um eine Datenbank zu konstruieren. Bei den in den einzelnen

Feldern gespeicherten Daten kann es sich um MATLAB-Variablen von einem beliebigen Datentyp handeln.

Strukturen kann man auf zwei verschiedene Arten erzeugen. Zum einen gibt es die Möglichkeit, eine Struktur direkt mit all ihren Variablen zu erzeugen:

```
>>strukturname=struct('Name1',Wert1,'Name2',Wert2,...);
```

Diese Eingabe hat den Vorteil, dass man alle Variablen der Struktur mit einem Befehl benennen und initialisieren kann.
Man erzeugt auch eine Struktur mit der Eingabe eines normalen Punktes. Wenn man also

```
>>raumkoordinate.x=1.5
```

eingibt, dann wird eine Struktur mit dem Namen `raumkoordinate` erzeugt, die bereits eine Variable `x` inklusive Wert enthält. Wenn man der Struktur `raumkoordinate` nun weitere Variablen (in diesem Fall sinnvoll wären natürlich die anderen beiden Koordinatenrichtungen) hinzufügen möchte, dann kann man dies mit dem gleichen Befehl tun. Um also die Koordinatenrichtungen zu komplettieren, gibt man beispielsweise

```
>>raumkoordinate.y=2.5; %Fügt raumkoordinate eine Variable y mit Wert 2.5 hinzu
>>raumkoordinate.z=3.2; %Fügt raumkoordinate eine Variable z mit Wert 3.2 hinzu
```

ein, und man erhält eine Struktur `raumkoordinate`, die die drei Koordinatenrichtungen abdeckt.

Man kann einer Variable einer Struktur auch eine Struktur zuweisen. Das kann im obigen Beispiel sehr sinnvoll sein. Denn jedes Objekt wird in Koordinaten haben. Also weist man sie ihm zu:

```
>>objektname.koordinate=raumkoordinate
```

Somit hat man nun eine Struktur, die eine weitere Struktur in sich eingebettet hat.

Um mit Hilfe von Strukturen Datenbanken zu erzeugen, kann man Strukturen auch in `Vektoren` speichern. Dafür wird dem Namen des Vektors dann einfach ein Index angehängt, wie man es von der normalen Vektorschreibweise gewohnt ist und anschließend mit dem Punkt und dem Variablennamen fortgesetzt. Der Zugriff auf die Variablen erfolgt dann also folgendermaßen:

```
>>array(index).variable=wert
```

So kann man mehrere Strukturen vom gleichen Typ in einem Vektor zusammenfassen. Der Index ist natürlich nur dann erforderlich, wenn mehr als eine Struktur in der Variable gespeichert werden soll. Wenn mehrere Strukturen vom gleichen Typ in dem Vektor gespeichert wurden und eine dieser Strukturen nun eine zusätzliche Eigenschaft erhalten soll, bekommen alle anderen Strukturen diese Eigenschaft auch. Allerdings ist bei diesen dann kein Wert eingetragen, d.h. die Variable ist leer.

Aufgaben und Übungen

3.39 ☞ *Struktur Auto*: Erstellen Sie eine einfache Struktur `auto`, die aus folgenden Dingen besteht: Farbe, Marke, PS-Zahl, Tankvolumen und Sitzanzahl. Benutzen Sie dazu zunächst die Methode mit dem Punkt. Setzen sie dabei beim Erstellen der einzelnen Variablen immer ein Semikolon

hinter die Eingabe, um die Ausgabe zu unterdrücken. Erst, wenn Sie komplett fertig mit der Eingabe aller Daten sind, geben Sie im Command Window `auto` ein und setzen diesmal kein Semikolon dahinter. Nun erstellen Sie eine zweite Struktur `auto2` mit der Funktion `struct` und setzen auch kein Semikolon dahinter.

3.40 ☞ *Datenbank*: Es soll eine Software geschrieben werden, um Datenbanken von Personen zu erzeugen. Schreiben Sie dafür ein Programm, bei dem Vorname, Nachname und Alter eingegeben und gespeichert werden können. Diese Information über die Personen soll als Vektor von Strukturen verwaltet werden. Das Programm soll auch in der Lage sein, bei der Eingabe eines Nachnamens die Information zu den entsprechenden Personen zu zeigen. Darüber hinaus soll es möglich sein, die vollständige Liste von Personen und Daten der Datenbank im Command Window zu zeigen. Die Datenbank muss in einer Datei `.\Datenbank\Liste.mat` gespeichert werden, damit sie beim nächsten Start des Programms geladen werden kann.

3.11 Probleme der numerischen Darstellung

Ein lineares Gleichungssystem $A \cdot x = b$ kann man anhand der Inversen der Matrix A lösen: Falls A regulär ist, gilt nämlich

$$A \cdot x = b \Rightarrow x = A^{-1} \cdot b$$

Wie man weiß, können reelle Zahlen nur mit endlicher Stellenzahl auf einem Rechner dargestellt werden. Sie werden also abgerundet. Dies geschieht auch mit dem Ergebnis jeder Operation von Zahlen, wenn das genaue Ergebnis nicht mehr innerhalb der Grenzen der Gleitpunktdarstellung passt.
Welche Wirkung dieser Fehler auf die Berechnung einer Matrizen-Inverse oder bei der Bestimmung der Lösung eines Gleichungssystems hat, soll hier anhand eines Beispiels angedeutet werden.

Geben Sie die Matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

ein und berechnen Sie deren Inverse mittels des Befehls `inv`. Berechnen Sie dann die Determinante dieser Matrix, indem Sie den Befehl `det` benutzen.

Was schließen Sie aus den Ergebnissen ?

Obwohl die Determinante der Matrix **A** Null ist (exakte Lösung), berechnet uns Matlab trotzdem eine Inverse für A. Um dieses Ergebnis weiter zu überprüfen, berechnen Sie das Produkt aus der Matrix und ihrer Inversen:

```
>> A * inv(A);
```

Das Ergebnis ist offensichtlich nicht gleich der Einheitsmatrix !!

Was ist die Erklärung dafür ? Der Algorithmus, welcher den Befehl `inv` aufruft, wird auf die Elemente der Matrix A eine Reihe von elementaren Operationen anwenden und gewisse Ergebnisse auswerten müssen, um die Aussage über die Invertierbarkeit der Matrix treffen zu können.

Gehen Sie zunächst davon aus, dass die folgenden Operationen durchgeführt werden würden

```
>>x = 0.1   ;
>>y = 1000  ;
>>a = 0.1   ;
>>b = y + a ;
>>c = b - y ;
>>d = c - a ;
```

und je nachdem, ob d verschwindet oder nicht, entscheidet der Algorithmus, ob die Matrix A singulär ist oder nicht. Wie man leicht nachrechnen kann, ist hier $d = 0$. Um nachzuprüfen, welches Ergebnis Matlab liefert, geben sie die oben angegebenen Operationen im Befehlsfenster ein.

Anscheinend ist d ungleich Null. Also würde Matlab an dieser Stelle den Algorithmus nicht abbrechen, sondern weiter die Inverse berechnen, was zu einem falschen Ergebnis führen würde.

Matlab hatte uns aber vorher gewarnt. Bei der Berechnung der Inversen von A erschien nämlich die Warnung

```
Warning: Matrix is close to singular or badly scaled.
 Results may be inaccurate. RCOND = 2.055969e-018.
```

Die zweite Aussage trifft mit Sicherheit in diesem Fall zu!
Was die erste bedeutet und wofür `RCOND` steht, soll anhand eines weiteren Beispiels [1] gezeigt werden.

Bestimmen Sie die Lösung folgenden Gleichungssystems:

$$A \cdot \mathbf{x} = b$$

mit

$$A = \begin{bmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{bmatrix} \qquad b = \begin{bmatrix} 32 \\ 23 \\ 33 \\ 31 \end{bmatrix}$$

Die exakte Lösung lautet x=$\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$.

Betrachtet wird nun die Empfindlichkeit der Lösung x gegenüber Störungen in den Koeffizienten von A und b. Dazu wird der relative Fehler einer Matrix y (mit der veränderten Matrix y_1) eingeführt:

$$er_y = \max\left(\left|\frac{y_{ij} - y_{1ij}}{y_{ij}}\right|\right)$$

Man wähle zunächst $b_1 = \begin{bmatrix} 32.1 & 22.9 & 33.1 & 30.9 \end{bmatrix}^T$. Berechen Sie den relativen Fehler von b. Bestimmen Sie die Lösung x_1 von $A \cdot x_1 = b_1$ (die exakte Lösung lautet: $x_1 = \begin{bmatrix} 9.2 & -12.6 & 4.5 & -1.1 \end{bmatrix}^T$) und den relativen Fehler von x. Wie lautet Ihr Fazit?

Als nächstes werden die Koeffizienten der Matrix A geringfügig verändert:

$$A_1 = \begin{bmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{bmatrix}$$

Berechnen Sie den relativen Fehler von A und die Lösung x_2 des Gleichungssystems $A_1 \cdot x = b$ (exakte Lösung: $\mathbf{x}_2 = \begin{bmatrix} -81 & 137 & -34 & 22 \end{bmatrix}^T$.
Berechnen Sie den relativen Fehler von $\mathbf{x}$.

Fazit: Kleine Änderungen in den Koeffizienten der Matrix A oder in b verursachen erhebliche Änderungen in der Lösung des Gleichungssystems $A \cdot x = b$. Man sagt, die Matrix A ist von schlechter Kondition und ein Maß dafür ist die Konditionszahl $\kappa(A)$, welche folgende Eigenschaft besitzt:

$$1 \leq \kappa(A)$$

Je näher diese Zahl bei 1 ist, desto besser ist die Matrix A konditioniert und man kann erwarten, dass MATLAB richtige Ergebnisse bei der Inversion oder bei der Lösung von Gleichungssystemen liefert.
Genau das Gegenteil ist der Fall, wenn die Konditionszahl sehr groß ist.

Ist diese sogar so groß, dass die Rechnergenauigkeit nicht mehr genügt, um sinnvolle Ergebnisse zu liefern, warnt MATLAB

[1] aus Larrouturou, Lions: Optimisation et commande optimale. Ecole Polytechnique

```
Warning: Matrix is close to singular or badly scaled.
 Results may be inaccurate. RCOND = 2.055969e-018.
```

wobei `RCOND` die Inverse der Konditionszahl von A ist, d.h. je näher `RCOND` an Null ist, desto schlechter ist die Matrix konditioniert.

Wären A und b das Ergebnis von vorher auf dem Rechner durchgeführten Kalkulationen, so hätten wir wahrscheinlich nicht die richtige Lösung getroffen! Hier liefert MATLAB die richtige Lösung, weil die Fehlerabweichung im Rahmen der Rechnergenauigkeit von MATLAB bleibt.

Was kann man unternehmen, um das Ergebnis zu verbessern?

1. Wenn man mit single precision arbeitet, sollte man auf jeden Fall die Genauigkeit in double precision (Standard bei MATLAB) umstellen.
2. Meistens benötigt man nicht explizit die Inverse einer Matrix. Zur Lösung des Gleichungssystems
$$A \cdot x = b$$
benutzt man dann anstatt `x = inv(A) · b` den Befehl der Linksdivision `A\ b` . Dieser beruht auf dem Gauss-Algorithmus, welcher keine Invertierung einer Matrix benötigt und generell stabil ist. Im Fall überbestimmter Gleichungssysteme liefert er die im Sinn der kleinsten Fehlerquadrate (least squares) beste Lösung.
3. Bei Matrizen, welche besondere Eigenschaften besitzen, z.B. symmetrisch oder positiv definit sind (u. a. der Fall bei Gleichungssystemen in der Parametrischen Identifikation von linearen zeitinvarianten Systeme), werden Algorithmen benutzt, welche diese Eigenschaft in Betracht ziehen (z. B. Cholesky Verfahren).

Fazit: Ziel dieses Abschnitts war nicht, Ihnen Angst vor dem Umgang mit MATLAB einzujagen. MATLAB liefert bei den meisten Berechnungen richtige Ergebnisse!
Das Ziel war vielmehr, auf gewisse Aspekte der numerischen Mathematik hinzuweisen, das Bewusstsein dafür zu schärfen und auch die Möglichkeit zu geben, mit MATLAB-Warnungen in dieser Richtung umzugehen.

Aufgaben und Übungen

3.41 ☞ Berechnen Sie die Lösung des Gleichungssystems

$$\begin{array}{llll} 1 \cdot x_1 & +2 \cdot x_2 & +3 \cdot x_3 & = 402 \\ 4 \cdot x_1 & +2 \cdot x_2 & +1 \cdot x_3 & = 521 \\ 7 \cdot x_1 & +5 \cdot x_2 & +9 \cdot x_3 & = 638 \end{array}$$

4 Einführung in Simulink

SIMULINK ist ein Programm zur numerischen Lösung linearer und nichtlinearer Differentialgleichungen, die das Verhalten physikalischer dynamischer Systeme durch ihre mathematischen Modelle beschreiben. Dazu besitzt SIMULINK eine graphische und blockorientierte Oberfläche, mit deren Hilfe die Gleichungen in Form von (Übertragungs-)Blöcken wie bei einem Wirkungsplan eingegeben und dargestellt werden.

4.1 Bedienoberfläche

Eine große Anzahl an vordefinierten Blöcken sind in sogenannten Bibliotheken zusammengefasst. Mit Hilfe der Maus können die Blöcke auf die Arbeitsfläche gezogen und anschließend parametriert werden.

Zuerst muß SIMULINK gestartet werden. Dies geschieht entweder durch Eingabe von `simulink` in dem MATLAB-Befehlsfenster oder durch Drücken des SIMULINK-Symbols in der Symbolleiste von MATLAB (Abbildung 4.1). Es öffnet sich der SIMULINK Library Browser, der den Zugriff auf die verschiedenen „Bibliotheken“ erlaubt. Die Bibliotheken sind in verschiedene Gruppen unterteilt: Continuous, Discrete etc.

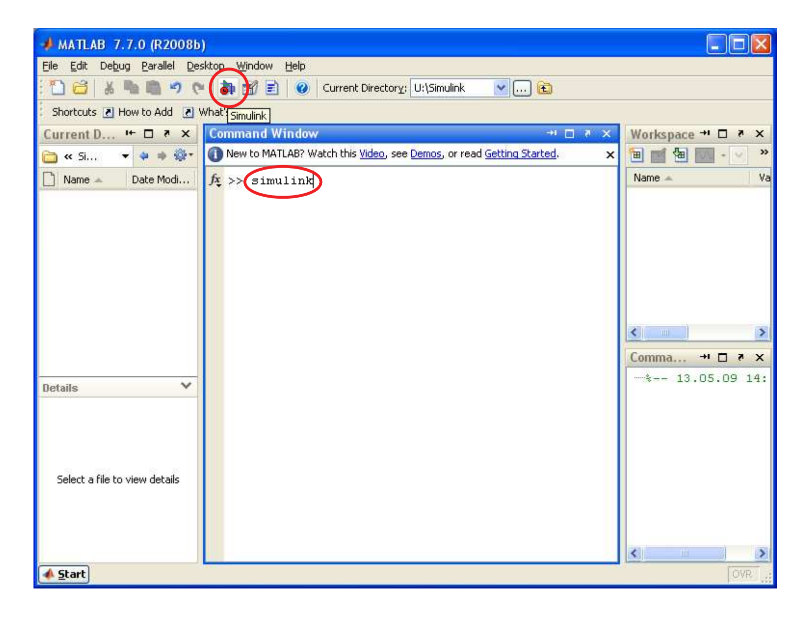


Abbildung 4.1: SIMULINK starten

Durch Drücken des Symbols *Create new model* (Abbildung 4.2) öffnet sich ein leeres Fenster ohne Namen (*untitled*). Der Name wird über *File* → *Save* beim Abspeichern eingegeben. SIMULINK-Modelle haben automatisch die Endung `.mdl`. Die mdl-Datei eines Modells dient zur Speicherung der Modellbestandteile, der Signalverbindungen und der Simulationsparameter. Eine mdl-Datei ist eine Textdatei und kann mit dem Editor verändert werden. Aus diesem Grund können mdl-Dateien auch mit Hilfe eines Versionsmanagements, wie beispielsweise Subversion oder Clearcase, verwaltet werden. Geöffnet wird eine Blockbibliothek durch einen Doppelklick auf ihr Symbol oder durch einen einfachen Klick auf das +. Mit der gedrückten linken Maustaste kann dann eine Kopie

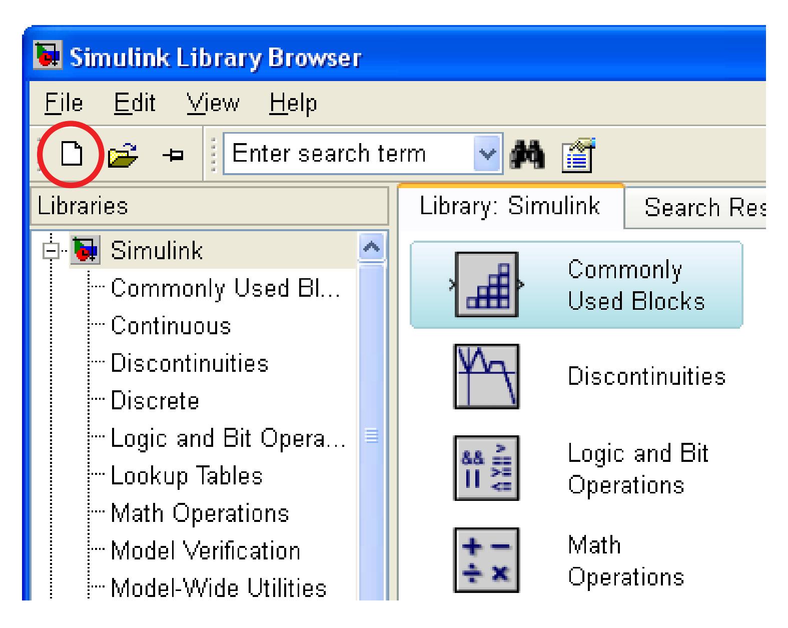


Abbildung 4.2: Neues Modell erstellen

des gewünschten Blocks auf das Modellfenster gezogen werden (Klicken & Ziehen). Sie können im Modellfenster mehrere Blöcke markieren, indem Sie beim Anklicken der einzelnen Blöcke die *Shift*-Taste gedrückt halten (Abbildung 4.3). Innerhalb des Modellfensters können

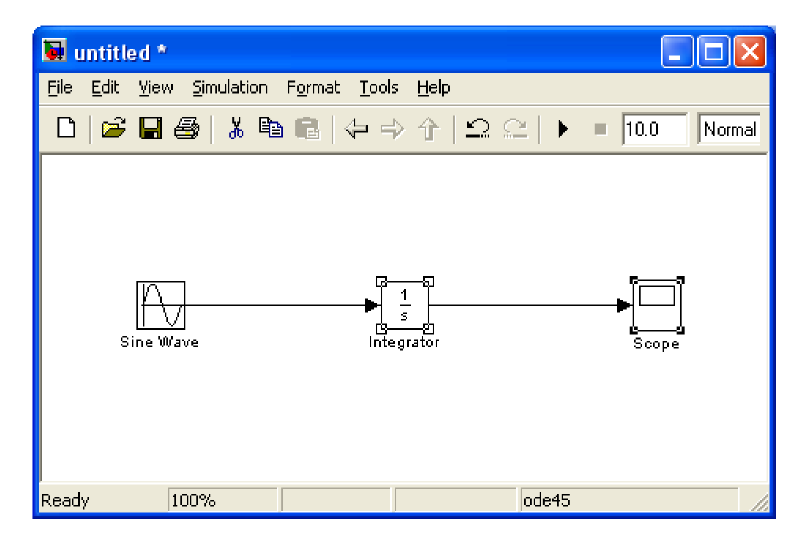


Abbildung 4.3: Einfaches Modell

- Blöcke durch Klicken & Ziehen mit der linken Maustaste *verschoben* werden,
- Blöcke durch Klicken & Ziehen mit der rechten Maustaste *kopiert* werden,
- Blöcke mit Hilfe der Menüpunkte *Format* → *Flip Block (bzw. Strg+I)* oder *Format* → *Rotate Block (bzw. Strg+R)* gedreht bzw. gespiegelt werden,
- **Signalverbindungen** durch Klicken & Ziehen mit der linken Maustaste von Blockausgängen zu Blockeingängen erzeugt werden,

- **Signalabzweigungen** durch Klicken & Ziehen mit der rechten Maustaste auf eine Signallinie erzeugt werden.
- Sie können das Simulink-Modell mit der Taste R hereinzoomen und mit der Taste V herauszoomen.

4.2 Simulink Standard-Block-Bibliothek

Simulink ist durch eine ganze Reihe Block-Bibliotheken erweiterbar, die für verschiedene Anwendungsgebiete entwickelt wurden (ähnlich wie die Toolboxen in Matlab). Im Rahmen dieser Kurzanleitung soll aber nur auf die Simulink Standard-Block-Bibliothek (Abb. 4.4) eingegangen werden, welche zusammen mit Simulink ausgeliefert wird.

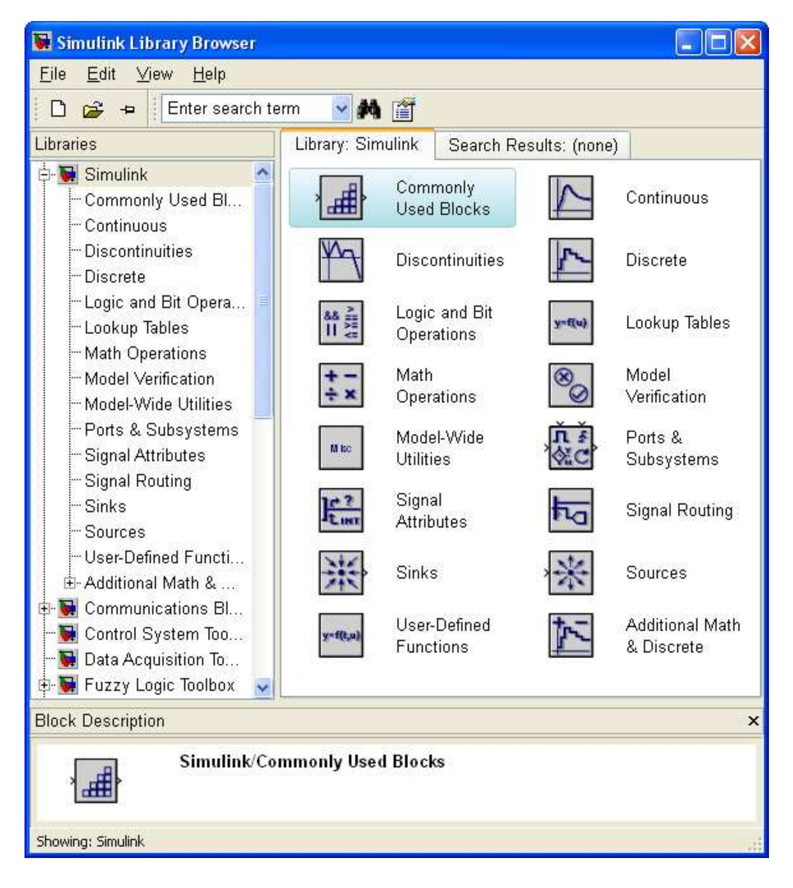


Abbildung 4.4: Simulink Standard-Block-Bibliothek

Die Standard-Block-Bibliothek ist zur besseren Übersicht in mehrere Gruppen unterteilt. Die wichtigsten Gruppen sind:

- *Commonly Used Blocks* enthält Elemente, die häufig gebraucht werden.
- *Continuous* enthält Elemente zur Darstellung linearer kontinuierlicher Systeme in Form der Übertragungsfunktion und im Zustandsraum.

- *Discontinuities* enthält nichtlineare Elemente zur Modellierung von Sättigungen (Saturation), Quantisierungen, Totzonen, Hysteresen etc.
- *Discrete* enthält Elemente zur Darstellung linearer zeitdiskreter Systeme in Form der Übertragungsfunktion und im Zustandsraum. Jeder diskrete SIMULINK-Block ist mit einem Abtaster und Halteglied 0. Ordnung ausgestattet.
- *Math Operations* enthält Basiselemente mathematischer Operationen (+, -, ∗, /, min/max, abs, sin, ...). Häufig benötigte Elemente sind *sum* und *gain* (Konstanter Faktor). Auch logische Operationen sind hier vorhanden.
- *Signal Routing* enthält Blöcke zum Zusammenfassen und Trennen von Signalen (z. B. *Mux*, *Demux*), Blöcke zur Erzeugung und Auswertung von Signalbussen, Schalter (*Switches*), Goto- und From-Blöcke etc.
- *Sinks* enthält Elemente zur Ausgabe, zum Darstellen und zum Speichern von Signalen. Z. B. können mit dem Block *To Workspace* Simulationsergebnisse in den MATLAB-Workspace geschrieben werden.
- *Sources* enthält Blöcke mit möglichen Eingangssignalen. Neben vordefinierten Signalformen (z. B. Step (Sprungfunktion), Constant) können u. a. auch Variablen aus dem Workspace mit beliebigen in Matlab erzeugten Werten als Eingangssignal angegeben werden (*From Workspace*).
- *User-Defined Functions* In dieser Bibliothek liegen Blöcke, mit denen eigene Funktionen realisiert werden können. (siehe auch Abschnitt 4.6).

Meistens werden andere Parameter als diejenigen benötigt, die standardmäßig bei den SIMULINK-Blöcken eingetragen sind. Zum Ändern der Parameter wird ein Block mit einem Doppelklick geöffnet. Hier ist auch die Hilfe (html) zu einzelnen Blöcken enthalten. Als Parameter können MATLAB-Variablen aus dem Workspace eingegeben werden. Sinnvollerweise werden alle in einem Modell benötigten Variablen entweder in einem mat-File gespeichert, das vor der Simulation geladen wird oder durch Ausführen eines m-Files erzeugt, das die entsprechenden Definitions-Befehle enthält.

Durch einen Doppelklick auf Verbindungen kann diesen ein Name gegeben werden. Der Name eines Blocks wird durch einfaches Anklicken (des Namens) und Editieren geändert. Das Menü *Format* ermöglicht, die Anzeige des Namens ein- und auszuschalten. Darüber hinaus können weitere Formate wie Zeichenfont, Farben, Schatten etc. eingestellt werden.

4.3 Scopes

Scopes dienen dem Beobachten und evtl. Speichern von Signalen. Standardmäßig enthält die Zeitachse (X-Achse) die Simulationsdauer und die Y-Achse den Bereich von -5 bis 5. Durch die Symbolleiste können die Einstellungen des Scopes verändert werden.

Die Symbolleiste (siehe Abb. 4.5) enthält die folgenden Funktionen von links nach rechts:

- *Drucken*
- *Parameter*. Dieses Symbol öffnet ein neues Dialogfenster.

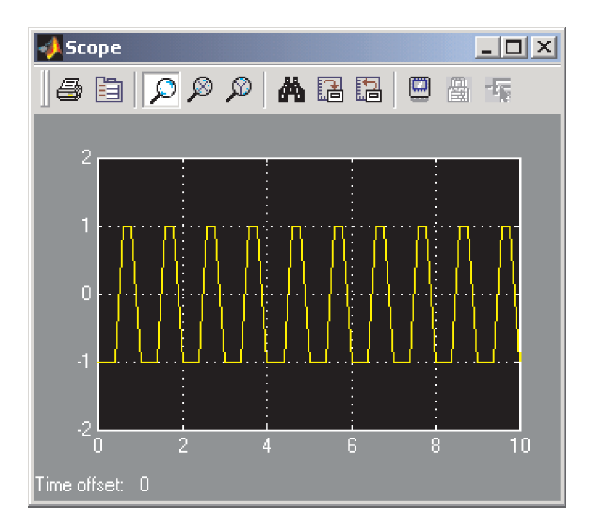


Abbildung 4.5: SIMULINK-Scope

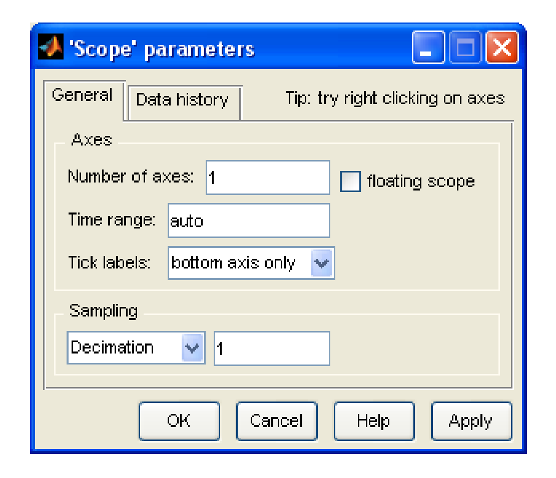


Abbildung 4.6: Scope Parameter General

– In der Karte **General** (Abbildung 4.6) können folgende Einstellungen getroffen werden:
  * **Number of axes** ändert die Anzahl der Subplots und dementsprechend die Anzahl der Eingänge. Die einzelnen Subplots besitzen die gleiche Zeitachse.
  * **Floating scope** erlaubt die Darstellung von Signalen, ohne sie mit dem Scope zu verbinden. Details können der MATLAB-Hilfe entnommen werden. Die Signale werden nur während der Simulationszeit angezeigt.
  * **Time Range** ermöglicht die Darstellung anderer Zeitbereiche als die Simulationsdauer (*auto*).
  * **Tick Labels** können hier ein- und ausgeschaltet werden.
  * **Sampling** stellt mit *Decimation* ein, jeder wievielte Wert im Scope dargestellt werden soll, wobei in dem Fall die Zeitintervalle nicht konstant sind. Mit *Sample Time* wird die Länge konstanter Zeitintervalle festgelegt.
– Die Karte **Data History** (Abbildung 4.7) erlaubt das Abspeichern der Simulationsergebnisse in einer Variablen (*Save Data to Workspace*) mit dem eingestellten Typ Struktur oder Array. Es kann sowohl die Zeit als auch die Daten gespeichert werden.

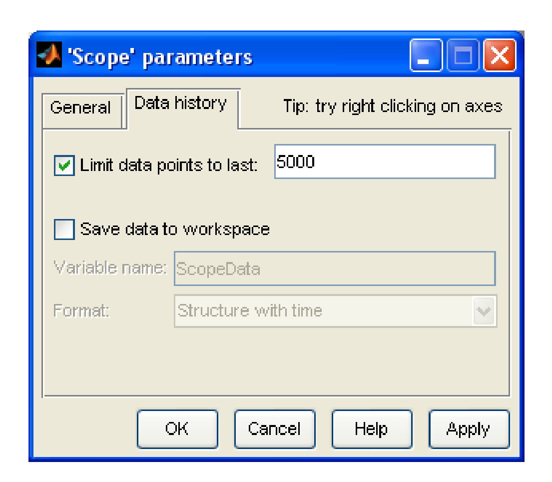


Abbildung 4.7: Scope Parameter Data History

Standardmäßig ist die Anzahl der im Scope dargestellten und zu speichernden Punkte auf 5000 begrenzt (*Limit data points to last*). Die Grenze kann verändert oder durch das Löschen der Markierung ganz aufgehoben werden.

- *Zoomen in X- und Y-Richtung.* Der zu vergrößernde Bereich wird durch das Ziehen mit gedrückter linker Maus-Taste markiert. Durch Drücken der rechten Maustaste wird die Vergrößerung wieder rückgängig gemacht.
- *Zoomen in X-Richtung*, Vergrößern und Verkleinern wie oben.
- *Zoomen in Y-Richtung*, Vergrößern und Verkleinern wie oben.
- *Autoscale.* Stellt die X- und Y-Achse so ein, dass der simulierte Graph gerade ganz sichtbar ist.
- *Save current axes settings.* Dieses Symbol ermöglicht das Speichern der aktuellen Achseneinstellungen als Standardzoom nach der Simulation.
- *Restore saved axes settings*: Zurückholen der gespeicherten Achseneinstellungen.
- Die letzten drei Symbole betreffen die FLOATING SCOPES. Details entnehmen Sie bitte der Hilfefunktion.

Andere Senken

Zwei weitere Möglichkeiten zum Abspeichern von Signalen sind mit den Blöcken *To Workspace* und *To File* vorhanden. Das Abspeichern mit *To Workspace* ist identisch zu *Save Data to Workspace* im Scope mit den gleichen Einstellungen. *To File* verlangt nach dem Datei- und Variablennamen, der Abtastzeit und speichert die Variable als Matrix im angegebenen File. Die Zeit wird jeweils mitgespeichert.

Display zeigt den aktuellen Wert des Signals numerisch an. *XYGraph* trägt ein Signal über einem anderen auf. *Stop Simulation* beendet die Simulation, sobald das Eingangssignal ungleich Null wird.

Aufgaben und Übungen

4.1 ☞ *Erste Schritte in* SIMULINK*:* Kopieren Sie einen Block *Transfer Fcn* aus der *Continuous*-Bibliothek. Speichern Sie das Modell als `test1.mdl` und schließen sie es. Sie können es durch Doppelklick auf die entsprechende Datei im Fenster *Current Directory*, oder durch Eingabe des Befehls `test1` im MATLAB-Befehlsfenster wieder laden.

Geben Sie als Eingangssignal auf die Übertragungsfunktion einen Sprung (engl. *step*) und stellen Sie das Ausgangssignal in einem Scope dar. Starten Sie die Simulation durch Drücken des Startsymbols ▷ in der Symbolleiste oder im Menü *Simulation* → *Start (bzw. Strg+T)* und überprüfen Sie das Ergebnis. Speichern Sie das Modell.

4.2 ☞ *Sinks:* Laden Sie erstmal das Modell, welches Sie in Aufgabe 4.1 erstellt haben.

- Geben Sie nun als Eingangssignal u eine Sinusschwingung $u = \sin(4t) + 2$ auf das System und sehen Sie sich das Ausgangssignal in einem Scope an.
- Stellen Sie mit Hilfe des *Mux*-Blocks Ein- und Ausgangssignal im gleichen Scope dar und ändern sie den Übertragungsfaktor der Übertragungsfunktion auf 3. Speichern Sie anschließend das Modell unter dem Namen `test2`.
- Ändern Sie das Eingangssignal zu einem Puls der Amplitude 3, der Periode 3s und der Breite von 1s und simulieren Sie das Ergebnis.

4.3 ☞ *Modell eines Autos:* Erzeugen Sie ein Modell eines Autos, das von einer Kraft $F_{Antrieb} = 3000N$ beschleunigt wird. Der Kraft $F_{Antrieb}$ wirken die Kräfte F_{Luft} und F_{Roll} für den Luft- und Rollwiderstand entgegen.

$$|\vec{F_{Luft}}| = C \cdot |\vec{v}|^2$$

Das Auto hat die Masse $M = 1500kg$ und einen Luftwiderstandskoeffizienten $C = 0.6\frac{kg}{m}$. Der Rollwiderstand F_{Roll} beträgt $250N$.
Geben Sie die Position und Geschwindigkeit des Autos mit Hilfe eines Scopes aus.

4.4 ☞ *Schwingkreise:* Erstellen Sie ein System, das das Modell eines der in Abbildung 4.8 dargestellten physikalischen Systeme enthält.

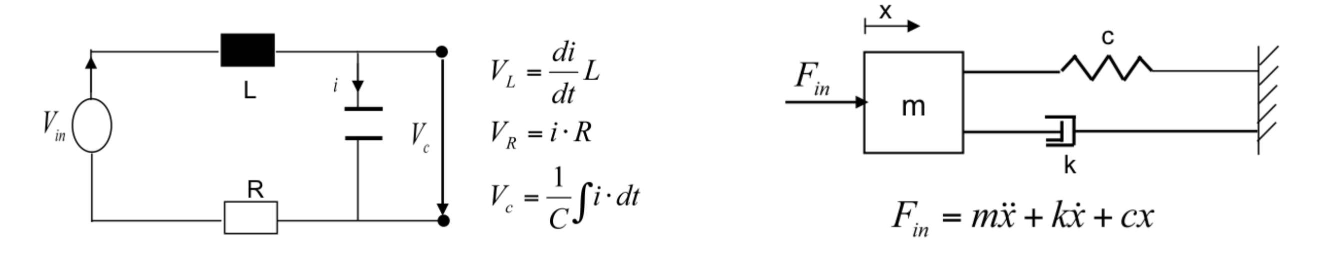


Abbildung 4.8: Schwingkreise

Simulieren Sie das System, indem Sie einen Sprung auf die Eingangsgröße in Höhe von $10V$ bzw. $2000N$ geben. Verwenden Sie als Parameter $R = 5\Omega, L = 2H$ und $C = 5mF$ bzw. $m = 20kg, k = 20\frac{kg}{s}$ und $c = 5000\frac{kg}{s^2}$.

4.4 Modellstrukturierung

Subsystems

Subsystems bieten die Möglichkeit der Hierarchisierung (mit beliebig vielen Ebenen), die für größere Modelle unumgänglich ist. Um ein Subsystem zu erstellen, werden die Blöcke, die zu einem Subsystem zusammengefasst werden sollen, markiert und der Menüpunkt *Edit* → *Create Subsystem* ausgewählt. Dieser Menüpunkt ist auch über das Kontextmenü (rechte Maustaste) erreichbar.

Durch einen Doppelklick werden Subsysteme innerhalb eines übergeordneten Modells geöffnet, jedoch nur wenn es sich um ein *unmaskiertes* Subsystem handelt. Im Falle eines maskierten Subsystems muss der Menüpunkt *Edit* → *Look Under Mask* ausgewählt werden. Mit einem Doppelklick auf ein maskiertes Subsystem wird die Paramtereingabemaske geöffnet, nicht aber das Subsystem selbst. Der Menüpunkt *Look Under Mask* ist ebenfalls wieder über das Kontextmenü erreichbar.

Sollen Signale an ein Subsystem gegeben oder von dem Subsystem erhalten werden, muss das Subsystem *Inport* bzw. *Outport*-Blöcke enthalten. Diese werden automatisch erzeugt.

Noch ein Hinweis: Die Blöcke *Enable* und *Trigger* aus der *Ports & Subsystems*-Bibliothek ermöglichen die bedingte Ausführung eines Subsystems. An dieser Stelle soll darauf nicht weiter eingegangen werden.

Maskieren von Subsystemen

Mit Hilfe des Menüs *Edit* → *Mask Subsystem* können Subsysteme maskiert werden, d. h. ihnen auf der übergeordneten Hierarchie ein bestimmtes Aussehen und Funktion gegeben, sowie ihre Inhalte versteckt werden. Es erscheint das Fenster *Mask Editor : SubSystemName* (Abb. 4.9) mit den vier Karten *Icon, Parameters, Initialization* und *Documentation.*

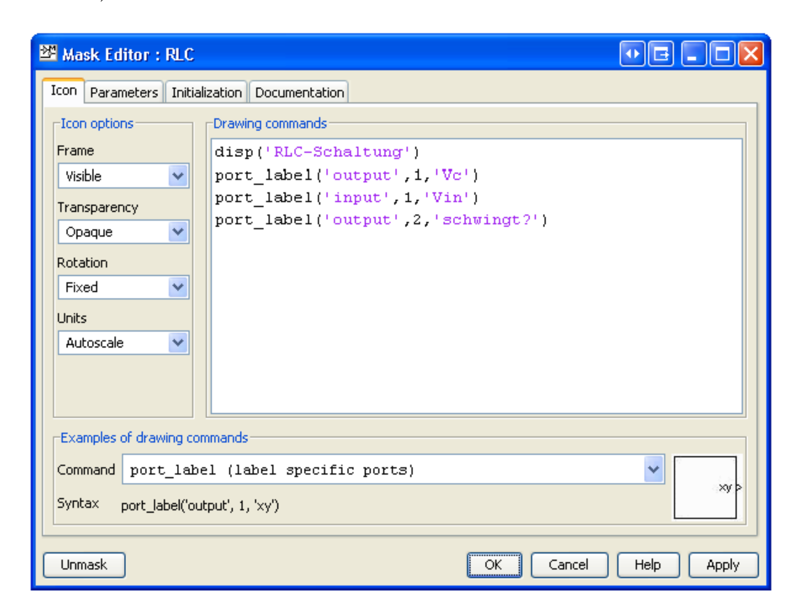


Abbildung 4.9: Mask Editor

- Icon
  erlaubt, ein eigenes Symbol für den Block festzulegen. Bei den **Drawing Commands** sind

alle Befehle zur Darstellung des neuen Symbols anzugeben. Text kann mit `disp`, Funktionsverläufe mit `plot` und Grafiken mit `image` gezeichnet werden.

- Parameters
  Die im Innern des Subsystems verwendeten Variablen müssen in der Variablenliste festgelegt werden. Ihre Namen werden im Feld *Variable* eingetragen, dazu passende Erläuterungen bei *Prompt*. Das Feld *Type* legt den Typ des Eingabefeldes fest (Editierfeld, Checkbox, Auswahlliste). Ist *Evaluate* markiert, wird der einzugebende Parameter von Matlab ausgewertet und falls möglich als Zahl übergeben, sonst als String. Ist *Tunable* markiert, können die Parameter zur Simulationslaufzeit verändert werden.

  Werden Variablen in dieser Liste als Parameter festgelegt, erscheint bei Doppelklick auf das Subystem für jeden Parameter das festgelegte Eingabefeld.

- Initialization
  erlaubt die Definition von lokalen Variablen und nötigen Initialisierungsberechnungen. Bei den *Initialization Commands* sind alle MATLAB-Befehle zulässig.

- Documentation
  erlaubt die Funktionalitäten des maskierten Blocks zu beschreiben und einen Hilfe-Text für diesen Block zu erstellen.

  Bei *Mask Type* kann der Maske ein Name gegeben werden. Der Text der unter *Mask Description* eingegeben wird, erscheint immer in der neuen Blockmaske. Unter *Mask Help* kann ein Hilfetext definiert werden. Die Hilfe wird automatisch im HTML-Format erstellt, so dass auch HTML-Befehle eingegeben werden können.

Hinweis: Eine Reihe von den SIMULINK Standardblöcken sind maskierte Subsysteme.

Aufgaben und Übungen

4.5 ☞ *Subsystem Auto:* Fassen Sie das Modell aus Aufgabe 4.3 in einem Subsystem zusammen. Die Eingangsgröße für das Subsystem ist die Kraft $F_{Antrieb}$, die Ausgangsgrößen sind die Position x und die Geschwindigkeit $\dot{x}$.
Maskieren Sie das Subsystem, sodass sich die Masse M, der Luftwiderstandskoeffizient C und der Rollwiderstand F_{Roll} des Autos über eine Eingabemaske parametrieren lassen.

4.6 ☞ *Maskierte Schwingkreise:* Maskieren Sie das Subsystem aus Aufgabe 4.4, sodass die verschiedenen Parameter wie m, k, L, etc. in der Maskenoberfläche eingegeben werden können. Geben Sie sinnvolle Hilfstexte und Erläuterungen. Simulieren Sie das System noch einmal und versuchen Sie es auch mit anderen Parametern.

4.7 ☞ *Maskiertes PT1-Glied:* Erstellen Sie aus einem Subsystem, das die Übertragungsfunktion eines PT1 enthält, einen maskierten Block mit dem Maskennamen PT1. Der Übertragungsfaktor und die Zeitkonstante sollen als Parameter eingegeben werden können. Das Symbol des Blocks soll die Übergangsfunktion enthalten. Geben Sie sinnvolle Erläuterungs- und Hilfetexte. Simulieren Sie mit diesem Block für verschiedene Kennwerte die Übergangsfunktion und die Antwort auf eine Sinus-Funktion.

4.5 Simulationsparameter

Die Simulationsparameter werden im Menü **Simulation** → **Configuration Parameters...** eingestellt. Es erscheinen unter anderem die Karten **Solver**, **Data Import/Export**und **Diagnostics**.

Solver

Die **Simulation time** legt fest, für welchen Zeitraum das Modell simuliert werden soll. Dies geschieht nicht in Echtzeit, sondern hängt davon ab, wie lange der Rechner für die Berechnungen benötigt. Die Startzeit kann dabei sowohl kleiner, gleich als auch größer Null sein. Wird an irgendeiner Stelle im Modell ein Startwert (**Initial Condition**) angegeben, so gilt diese für den festgelegten Startzeitpunkt der Simulation und nicht automatisch für Null. Die **Simulation stop time** kann entweder in dem entsprechenden Feld bei **Simulation time** eingeben werden, oder direkt in der Toolbar, rechts neben dem Stop-Zeichen.

Simulieren bedeutet, dass das in Blöcken erstellte Differentialgleichungssystem mit Hilfe numerischer Integrationsverfahren schrittweise gelöst wird. Bei den **Solver options** wird der gewünschte Integrationsalgorithmus ausgewählt, festgelegt, ob mit oder ohne Schrittweitensteuerung gearbeitet wird (**Fixed-step** oder **Variable-step**), und Werte für die maximale Schrittweite (**Max step size**), die Anfangsschrittweite (**Initial step size**) sowie die Fehlertoleranzen der Schrittweitensteuerung festgelegt.

Es wird empfohlen, als Integrationsalgorithmus für kontinuierliche Systeme **ode45** und für diskrete Systeme **discrete** einzustellen. Die anderen Möglichkeiten erfordern eine bessere Kenntnis der numerischen Mathematik.

Data Import/Export

Anstatt mit dem (Source-)Block **Load from workspace** kann auf Signale aus dem Workspace auch mit dem Block **In1** (Input Port) aus der **Signals & Systems** Bibliothek zugegriffen werden, sobald ihre Variablennamen im Feld **Load from workspace** eingegeben worden sind. Die erste Spalte der eingegebenen Matrix wird als Zeitvektor genommen.

Save to workspace funktioniert analog mit dem **Out1** (Output Port) und stellt daher eine weitere Möglichkeit dar, Simulationsdaten im Matlab-Workspace anzulegen. Die **Save options** sind ebenfalls analog zu den Einstellungen in den Scopes und den To Workspace-Blöcken.

Mit den **Output options** (nur verfügbar bei variabler Schrittweite, siehe Solver) können zusätzliche Punkte in einem Integrationsintervall der Ausgabe hinzugefügt werden. Der **Refine factor** gibt an, in wie viele kleinere Intervalle ein Integrationsintervall mit Hilfe der Interpolation für die Ausgabe aufgeteilt werden soll. **Produce additional output** zwingt den Integrationsalgorithmus Schritte zu den angegebenen Zeiten zusätzlich zu berechnen. Im Gegensatz dazu wird mit **Produce specified output only** nur zu den angegebenen Zeitpunkten (mit Start- und Endzeit) ein Integrationsschritt durchgeführt.

Diagnostics

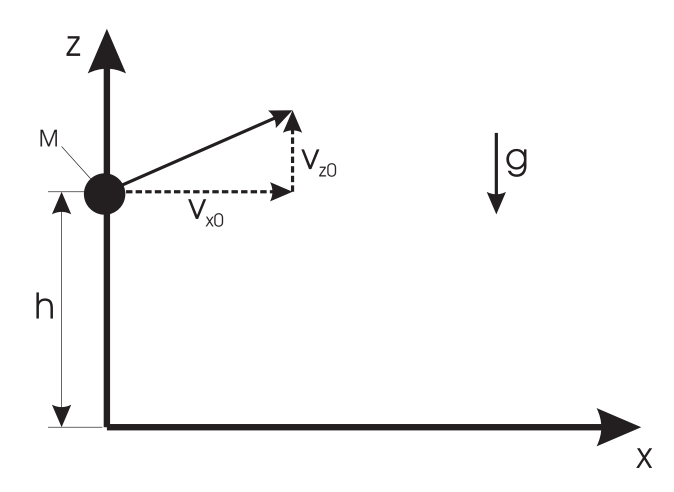
Auf dieser Karte kann man auswählen, bei welchen Ereignissen, welche Art von Fehlermeldungen ausgegeben werden sollen: keine (**none**), Warnung (**warning**) oder Fehler (**error**). Im Allgemeinen

sollten die Standardeinstellungen bis auf wenige Ausnahmen nicht verändert werden müssen.

Eine sehr wichtige Meldung ist diejenige eines **Algebraic loop**. Eine sogenannte algebraische Schleife tritt dann auf, wenn ausschließlich Blöcke mit Durchgriff einen geschlossenen Wirkungskreis bilden. Bei Blöcken mit Durchgriff hängt das aktuelle Ausgangssignal vom aktuellen Eingassignal ohne Verzögerung ab, z. B. beim Gain. Die Rechenzeit wird durch eine algebraische Schleife stark verlangsamt. Durch Einfügen eines **Memory**-Block aus der Continuous-Bibliothek lässt sich die Schleife aufbrechen. Eine weitere Möglichkeit liegt in der Verwendung des **Algebraic Constraint**-Blocks aus der Math Operations-Bibliothek zur Lösung algebraischer Gleichungs-/Differentialgleichungssysteme.

Aufgaben und Übungen

4.8 ☞ *Schiefer Wurf:* Simulieren Sie die Flugbahn einer punktförmigen Masse M in der (x, z)-Ebene, welche im Punkt $(0, h)^T$ mit der Geschwindigkeit $(v_{x0}, v_{z0})^T$ losgeworfen wird:



Vernachlässigen Sie zunächst alle Reibungs- und Kontakteffekte.
Es gilt: $M = 2\text{kg}$, $v_{x0} = v_{z0} = 10\frac{\text{m}}{\text{s}}$, $h = 5\text{m}$, $g = 9.81\frac{\text{m}}{\text{s}^2}$.
Tipp: Es gilt

$$\begin{aligned} \dot{x} &= v_{x0} \\ \ddot{z} &= -g \end{aligned}$$

- Setzen Sie diese beide Gleichungen mit Hilfe von Integrierern in Simulink um. Vergessen Sie nicht die Anfangsbedingungen. Speichern Sie das System unter dem Namen `SchieferWurf.mdl`.
- Simulieren Sie das System für eine Zeitdauer von $t_{sim} = 5\text{s}$. Stellen Sie die Zeitverläufe von $x(t)$, $\dot{x}(t)$, $z(t)$ und $\dot{z}(t)$ in einem Scope und $x(t)$ und $z(t)$ in einem x-y-Plot dar.

In diesem einfachsten Fall ist neben der numerischen Lösung des Differentialgleichungssystems (Simulink) natürlich auch eine analytische Lösung möglich.

- Integrieren Sie das System per Hand und ermitteln Sie die analytische Lösung. Stellen Sie die Lösungen für $x(t)$ und $z(t)$ in einem weiteren Scope dar und vergleichen Sie die beiden Lösungen.
  Tipp: Sie benötigen den Clock-Block aus der Sources-Bibliothek.

4.9 ☞ *Schiefer Wurf mit Reibung:* Öffnen Sie das Modell `SchiferWurf.mdl` aus Aufgabe 4.8. Im folgenden sollen Reibungs- und Kontakteffekte in die Simulation integriert werden, so dass die Lösung der DGL nunmehr numerisch erfolgen kann.

- Ergänzen Sie die Simulation um einen „Boden" $z = 0$. Modellieren Sie ihn in der Weise, dass er sich in z-Richtung wie ein Feder-Dämpfer-Glied verhält. Kommt also die Punktmasse mit dem Boden in Kontakt, überträgt dieser folgende Kraft auf die Punktmasse (in z-Richtung):

  $$F_{zB} = -k_b z - k_d \dot{z} \quad \text{für } z < 0$$

  mit $k_b = 1\frac{\text{kN}}{\text{m}}$, $k_d = 10\frac{\text{Ns}}{\text{m}}$.
  Simulieren Sie nun für eine Zeitdauer von $t_{sim} = 10\text{s}$.

- Ergänzen Sie nun den Bodenkontakt um eine Reibkomponente in x-Richtung. Im Kontaktfall wird gemäß dem Reibungsgesetz in x-Richtung folgender Betrag der Reibkraft auf den Massepunkt übertragen:

  $$|F_R| = \mu |F_N| \quad \text{für } z < 0,$$

  wobei $\mu = 0.1$. F_N ist die Normalkraft, mit welcher der Massepunkt auf den Boden drückt. Überlegen Sie sich selbst die Richtung der Kraftwirkung.

- Ergänzen Sie das Modell um Luftreibung. Der Betrag der Luftreibungskraft ergibt sich zu

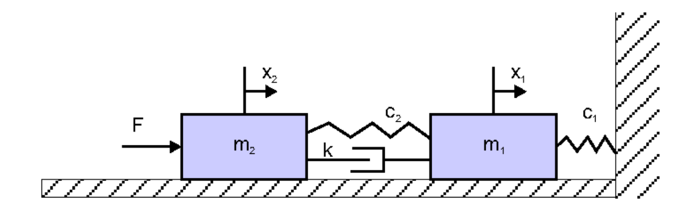
  $$|\vec{F}_L| = c_L |\vec{v}|^2 = c_L \left| \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \right|^2,$$

  wobei $c_L = 0.01\frac{\text{Ns}^2}{\text{m}^2}$. Ihre Richtung weist entgegen dem Geschwindigkeitsvektor $\vec{v} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}$.

Wie weit kommt der Massepunkt in x-Richtung in Ihrer Simulation? In welcher z-Position kommt er zu Ruhe? Ab welchem Zeitpunkt kommt die Simulation nicht mehr weiter? Weshalb? Wie kann man dieses Problem umgehen?

4.10 ☞ *Speichern von Simulink-Variablen:* Laden Sie das Modell aus Aufgabe 4.9. Speichern Sie den simulierten Verlauf von $x(t)$ als Matrix im Workspace unter dem Variablennamen `xt`. Speichern Sie den Verlauf von $z(t)$ in einer Datei `testdat.mat` unter dem Variablennamen `zt`. Kontrollieren Sie das Ergebnis im Workspace und in der Datei. Ändern Sie den Decimation Factor auf 10 und 100 und betrachten Sie das Ergebnis.

4.11 ☞ *Zweimassenschwinger:* Gegeben ist folgendes mechanische System mit $m_1 = 1, m_2 = 5, c_1 = 3, c_2 = 1, k = 0.5$:



- Stellen Sie die Differentialgleichungen der Bewegung der beiden Massen mit dem Eingangssignal F auf.

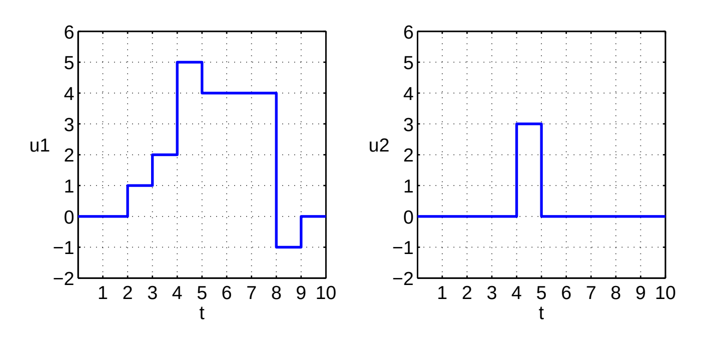
- Erstellen Sie ein Blockschaltbild, das neben Elementen aus den Blockbibliotheken Quellen und Senken ausschließlich die Blöcke vom Typ **Gain**, **Integrator** und **Sum** enthält. Speichern Sie das Modell als **Test3**.
- Simulieren sie nun das System über eine Zeitdauer von 30 Sekunden, falls ein Einheitskraftsprung auf das System aufgebracht wird. Sind die Ergebnisse plausibel?
- Ändern Sie nun den Refine Factor auf 10 und vergleichen Sie die Simulationsergebnisse. Ändern Sie die Fehlertoleranzen sowohl zu genaueren als auch ungenaueren Werten. Vergleichen Sie die Simulationsergebnisse.

Hinweis: Schreiben Sie das System in Zustandsraumform, damit Sie die Gleichungen als Blockschaltbild darstellen können. Wählen Sie dafür $z_1 = x_1, z_2 = \dot{x}_1, z_3 = x_2, z_4 = \dot{x}_2$.

4.12 ☞ *Zeitkontinuierliches Zustandsraummodell:* Simulieren Sie folgendes Differentialgleichungssystem mit einem Zustandsraummodell:

$$\dot{\mathbf{x}} = \begin{bmatrix} -0.2667 & -0.3333 \\ 1 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & -2 \\ 0 & 0.5 \end{bmatrix} \mathbf{u}, \quad \mathbf{y} = \begin{bmatrix} 1.67 & 0 \\ 0 & 1.2 \\ 1 & 1 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \mathbf{u}$$

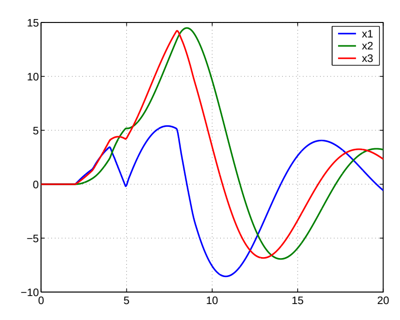
Simulieren Sie das System mit ode45 und variabler Schrittweite für die Dauer von 20 Sekunden, während am Eingang folgende Signale anliegen. Geben Sie **y** auf einem Scope aus.



Tip: Definieren Sie im MATLAB-Workspace die Vektoren

```
u1 = [0 0 1 2 5 4 4 4 -1 0 0]'
u2 = [0 0 0 0 3 0 0 0 0 0 0]'
t = [0:10]'
```

und benutzen Sie in Simulink den `From-Workspace`-Block. Es soll sich folgender *Verlauf* der Zustände ergeben:



4.6 User-defined Functions Bibliothek

SIMULINK bietet die Möglichkeit benutzerdefinierte Funktionen in ein Modell einzubinden, die sich mit den vordefinierten Blöcken nicht, oder nur sehr aufwändig realisieren lassen würden. Um beispielsweise die Funktion $f(x) = e^{-x} \cdot sin(x) + 1$ mit vordefinierten SIMULINK-Blöcken darzustellen sind bereits sechs Blöcke nötig.

Fcn und MATLAB-Fcn

Der `Fcn`-Block wendet den angegebenen mathematischen Ausdruck auf den Eingang an und gibt das Ergebnis aus. Mit Hilfe des `Mux`-Blocks können auch vektorielle Eingänge verarbeitet werde, der Ausgang ist jedoch immer skalar. Es können folgende Ausdrücke verwendet werden:

- u ist die Eingangsvariable. Bei vektoriellem Eingang kann mit `u(i)` auf das i-te Element zugegriffen werden.
- Numerische Konstanten wie z.B. 1, 2, 3 etc.
- Arithmetische Operationen: + - * / ˆ
- Relationale Operatoren: == (gleich), != (ungleich), >(größer), <(kleiner), >= (größer oder gleich) und <= (kleiner oder gleich). Das Ergebnis dieser Operatoren ist entweder 1, falls die Bedingung zutrifft, ansonsten 0.
- Logische Operatoren: && (und), ||(oder) und ! (nicht). Das Ergebnis dieser Operatoren ist entweder 1, falls die Bedingung zutrifft, ansonsten 0.
- Mathematische Funktionen: abs, acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, hypot, ln, log, log10, pow, power, rem, sgn, sin, sinh, sqrt, tan, und tanh.
- Workspace Variablen: Namen, die in der vorangegangenen Liste nicht auftauchen, werden in MATLAB ausgewertet. Es können nur skalare Variablen benutzt werden, bzw. muss z.B. mit `A(1,2)` eine einzelne Komponente einer Matrix angegeben werden.

Mit dem `MATLAB-Fcn`-Block können Funktionen aus MATLAB auf den Eingang angewendet werden. Dabei können sowohl vordefinierte als auch selbst geschriebene MATLAB-Funktionen verwendet werden. Die Rückgabe der Funktion muss mit der Dimension des Blocks übereinstimmen, sonst führt dies zu einem Fehler. Als Eingang werden reale oder komplexe Werte vom Typ *double* akzeptiert. Bei vektoriellen Eingangsgrößen kann wieder wie beim *Fcn*-Block mit Hilfe von Indizes auf einzelne Komponenten zugegriffen werden.

`Fcn`-Blöcke können sehr viel schneller als `MATLAB-Fcn`-Blöcke ausgeführt werden, da letztere bei jeder Berechnung erst durch MATLAB interpretiert werden müssen. Die `MATLAB-Fcn`-Blöcke müssen die m-File während der Laufzeit mehrmals aufrufen und sind deswegen sehr langsam. Es empfiehlt sich an der Stelle `Embedded-MATLAB-Fcn` (s. u.) zu verwenden, da diese kompiliert und in das Modell eingebettet werden.

Aufgaben und Übungen

4.13 ☞ *Anwendung von Fcn:* Erstellen Sie mit Hilfe vom `Fcn`-Block ein Modell, das die Funktion

$$y(t) = \begin{cases} 0 & \forall t \leq 1sec \\ \sin(t) \cdot u(t) & \forall t > 1sec \end{cases}$$

abbildet. Als Eingang $u(t)$ nehmen Sie ein `Chirp Signal` mit den Standardparametern.

4.14 ☞ *Steinfall:* Ein Stein fällt aus einer unbekannten Höhe. Es kann gezeigt werden, dass folgende Gleichung die Geschwindigkeit über der Zeit darstellt (Anfangsgeschwindigkeit ist hier Null):

$$v(t) = \tanh\left(\frac{t\sqrt{mgk}}{m}\right)\frac{\sqrt{mgk}}{k}$$

wobei $m = 1kg$, $k = 1\frac{kg}{m}$ und $g = 9,81\frac{m}{s^2}$ gilt. Erstellen Sie das (analytische) Modell in Simulink mit `stop time=inf` und halten Sie die Simulation (Block STOP), wenn der Stein seine Endgeschwindigkeit erreicht hat. Geben Sie den Verlauf der Geschwindigkeit mit einem Scope aus.
Tipp: Um zu wissen, ob die Endgeschwindigkeit erreicht wurde, können Sie $v(t)$ zeitlich ableiten und mit einem `Fcn`-Block überprüfen, ob der Betrag sehr klein ist (z.B. 10^{-3}).

Eingebettete Matlab-Funktionen

Mit Hilfe des `Embedded-MATLAB-Function`-Block lassen sich MATLAB-Funktionen direkt in SIMULINK einbetten, d.h. es wird nicht einfach eine Funktion aus MATLAB benutzt, die beispielsweise als m-File vorliegt, sondern die Funktion wird in das SIMULINK-Modell integriert und mit dem Modell kompiliert, was eine sehr viel schnellere Ausführung ermöglicht. Die Anzahl der Ein- und Ausgänge des Blocks wird dabei durch die Funktion bestimmt, es sind also auch mehrere Ein- und Ausgänge möglich.

Die Syntax von diesen Funktionen sieht folgendermaßen aus:

```
function [y1 y2] = Funktionsname(u1,u2,p1,p2)
% This block supports the Embedded MATLAB subset.
% See the help menu for details.
```

Hier sind y_i Ausgangsgrößen, u_i Eingangsgrößen und p_i Parameter. Die Ein-/Ausgangsgrößen können auch vektorielle Form haben. Somit lassen sich sehr viele statische Systeme darstellen, auch Systeme mit nicht-linearem Verhalten. Die eingebetteten Matlab-Funktionen sind aber nicht geeignet, dynamische Systemen abzubilden. Falls das zu modellierende System dynamische Eigenschaften *(Differentialgleichung)* aufweist, müssen z.B. `s-Function`-Blöcke angewendet werden.

Die Parameter werden wie Eingangsgrößen nach dem Funktionsnamen definiert und müssen nachher im *Embedded MATLAB Editor* (s. Abbildung 4.10) als solche gekennzeichnet werden. Zur Simulation können später diese Parameter direkt aus dem Workspace gelesen werden oder auch über eine Maske vom Nutzer eingegeben werden.

Nicht alle in Matlab möglichen Funktionen und Variablentypen sind in Embedded MATLAB Functions erlaubt. In der Matlab-Hilfe kann eine Liste mit den unzulässigen Konstrukten (unsupported features) gefunden werden. Beispielsweise dürfen in eingebetteten Funktionen keine *globalen* Variablen und keine *Cell Arrays* verwendet werden. Außerdem ist die dynamische Speicherzuweisung (wie Vektoren ohne feste Dimension) nicht erlaubt.

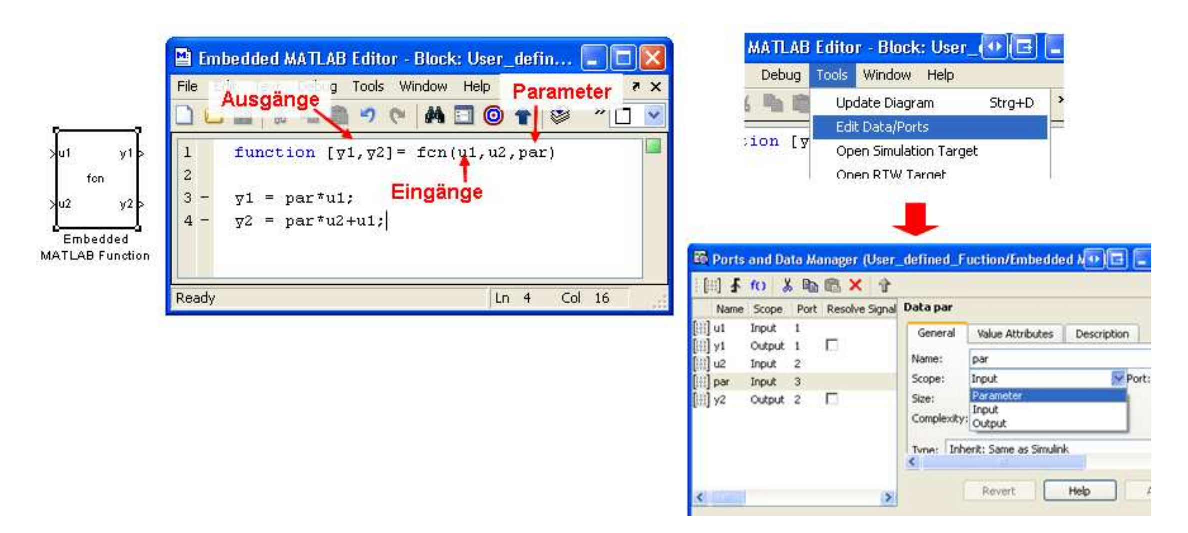


Abbildung 4.10: Embedded MATLAB Editor, Parametereinstellung

Aufgaben und Übungen

4.15 ☞ *Anwendung von Embedded MATLAB Fcn:* Wiederholen Sie die Aufgabe 4.13, aber verwenden Sie statt des Blocks `Fcn` eine eingebettete Matlab-Funktion.

4.16 ☞ *Steinfall mit Parametern:* Wiederholen Sie die Aufgabe 4.14, aber verwenden Sie statt des Blocks `Fcn` eine eingebettete Matlab-Funktion. Stellen Sie m und k als Parameter der eingebetteten Funktion ein.

4.17 ☞ *Transformation zu Polarkoordinaten:* Erstellen Sie eine eingebettete Funktion, welche die zwei Komponenten eines Vektors (x, y) in kartesischen Koordinaten als Eingangsargumente bekommt und die entsprechenden Polarkoordinaten $(\rho, \phi) = \left(\sqrt{x^2 + y^2}, atan(\frac{y}{x})\right)$ erzeugt.

S-Functions (alt)

S-Functions sind MATLAB-Funktionen (oder C oder Fortran, was hier nicht weiter behandelt werden soll) mit festgelegter Parameterstruktur, mit deren Hilfe beliebige lineare und nichtlineare Differentialgleichungs-Systeme in Zustandsraumdarstellung modelliert werden können.

Der Aufruf von SIMULINK erfolgt über die Verwendung des Blocks **S-Function** aus der Bibliothek **User-Defined Functions**. In der Eingabemaske wird der Funktionsname der zu verwendenden S-Function sowie bei Bedarf die zu übergebenden Parameter eingegeben. Eine S-Function kann maskiert werden, sodass Erläuterungen zu den eingegebenen Parametern und der Wirkungsweise des Blockes hinzugefügt werden können.

Der Funktionskopf einer S-Function besitzt folgendes Aussehen:

```
[sys, x0, str, ts] = sfunctionname(t, x, u, flag);
```

oder bei Eingabe von zusätzlichen Parametern p1 bis pn:

```
[sys, x0, str, ts] = sfunctionname(t, x, u, flag, p1, ... , pn);
```

Mit dem Parameter **flag** wird bestimmt, was die S-Funktion beim aktuellen Aufruf aus der Zeit t, dem aktuellen Zustand x und dem aktuellen Eingangssignal u berechnet. Die Steuerung des

Aufrufs und damit des Parameters flag liegt vollständig intern bei SIMULINK.

- flag = 0: Initialisierung
  Erfolgt einmalig zu Simulationsbeginn. Die Variable sys muss ein Vektor mit 7 ganzzahligen Werten sein, wobei die Werte z. B. die Anzahl an Ein- und Ausgängen und Zuständen festlegen.
- flag = 1: Berechnung der Ableitung kontinuierlicher Zustandsgrößen
  Erfolgt bei jedem Simulationsschritt. Hier wird die kontinuierliche Zustands-Differentialgleichung

  sys = f(x, u) (% $=\dot{x}$)

  programmiert. sys ist jetzt ein Vektor mit so vielen Einträgen wie das System kontinuierliche Zustände besitzt. Dies wurde bei der Initialisierung festgelegt.
- flag = 2: Aktualisierung diskreter Zustände
  Erfolgt bei jedem Simulationsschritt. Hier wird die diskrete Zustands-Differenzengleichung

  sys = f(x_k, u_k) (% $=x_{k+1}$)

  programmiert. sys ist jetzt ein Vektor mit so vielen Einträgen wie das System diskrete Zustände besitzt. Dies wurde bei der Initialisierung festgelegt.
- flag = 3: Berechnung der Ausgangsgrößen
  Erfolgt bei jedem Simulationsschritt. Hier wird die Ausgangsgleichung

  sys = g(x, u) (% = y)

  programmiert. sys ist jetzt ein Vektor mit so vielen Einträgen wie das System Ausgänge besitzt. Dies wurde bei der Initialisierung festgelegt.
- flag = 9: Abschließende Tasks
  Erfolgt einmal zu Simulationsende. Hier werden alle abschließenden Befehle eingegeben, z. B. Variablen aus dem Workspace löschen.

Eine ausführliche Erläuterung finden Sie in der Datei **sfuntmpl.m**, die als Vorlage für selbsterstellte S-Functions dient. Eine einfache S-Function ist die Datei **sfuncont.m**, die einen Integrierer darstellt. Beide Dateien sind im Verzeichnis `<MATLAB> \toolbox\simulink\blocks` enthalten.

S-Functions

S-Functions sind MATLAB-Funktionen (oder C-/C++-/Fortran-Funktionen, was hier nicht weiter behandelt werden soll) mit festgelegter Parameterstruktur, mit deren Hilfe beliebige lineare und nichtlineare Differentialgleichungs-Systeme in Zustandsraumdarstellung modelliert werden können.

M-file S-Functions werden über den Block **Level-2 M-file S-Function**[1] aus der Bibliothek **User-Defined Functions** im *Simulink Library Browser* eingefügt. In der Eingabemaske wird der Funktionsname der zu verwendenden S-Function sowie bei Bedarf die zu übergebenden Parameter eingegeben. Eine S-Function kann maskiert werden, sodass Erläuterungen zu den eingegebenen Parametern und der Wirkungsweise des Blockes hinzugefügt werden können.

[1] Der Zusatz *Level-2* lässt darauf schließen, dass es auch *Level-1 M-file S-Functions* gibt. Diese stammen aus einer älteren MATLAB-Version. Sie werden zwar noch unterstützt, jedoch wird in der MATLAB-Hilfe dazu geraten, für neue *M-file S-Functions* die Level-2-API zu benutzen.

Eine M-file S-Function muss mindestens die folgenden Methoden enthalten:

- eine *setup*-Funktion und
- eine *Output*-Funktion

In der setup-Funktion werden die Anzahl der Eingangs- und der Ausgangsgrößen sowie deren Attribute (Dimensionen, Datentypen, Sample times etc.), die Sample Time des Blocks, die Anzahl der Parameter im S-Function-Dialog und Ähnliches festgelegt.

In der Output-Funktion wird/werden die Ausgangsgröße(n) berechnet.

Außer den beiden genannten sowie eventuell weiteren benötigten Methoden enthält die S-Function nichts anderes als einen Aufruf der setup-Funktion.

Ein einfaches Beispiel für eine S-Function ist die Datei **msfcn_times_two.m** im Verzeichnis <MATLAB> `\toolbox\simulink\simdemos\simfeatures\`. Diese S-Function verdoppelt das Eingangssignal, entspricht also einem *Gain*-Block mit einem Verstärkungsfaktor von 2.

In der setup-Funktion werden zunächst die Anzahl der Eingangs- und die der Ausgangsgrößen jeweils auf 1 gesetzt:

```
function setup(block)

  %% Register number of input and output ports
  block.NumInputPorts  = 1;
  block.NumOutputPorts = 1;
```

Die beiden folgenden Funktionsaufrufe sorgen dafür, dass Eingangs- und Ausgangsport ihre Eigenschaften (Dimensionen, Datentypen, Komplexität und Sampling-Modus) vom Modell übernehmen:

```
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;
```

Das nachfolgende Schritt ist immer dann nötig, wenn das Ausgangssignal eine unmittelbare Funktion vom Eingangssignal ist:

```
  block.InputPort(1).DirectFeedthrough = true;
```

Als nächstes wird festgelegt, dass die Sample Time vom Eingangssignal übernommen wird:

```
  block.SampleTimes = [-1 0];
```

Die nächsten beiden Funktionen können sind für dieses Beispiel nicht notwendig und werden nicht weiter beachtet.

Der letzte Funktionsaufruf innerhalb der setup-Funktion ist jedoch untentbehrlich. Hier wird festgelegt, dass die Berechnung des Ausgangssignals in der Funktion *Output* zu finden ist:

```
  block.RegBlockMethod('Outputs', @Output);
```

Die Output-Funktion enthält nur eine Zeile, in der das Ausgangssignal aus dem Eingangssignal berechnet wird:

```
function Output(block)

   block.OutputPort(1).Data = 2*block.InputPort(1).Data;
```

Als Vorlage für eigene S-Functions können Sie die Datei **msfuntmpl.m** oder **msfuntmpl_basic.m** aus dem Verzeichnis `<MATLAB> \toolbox\simulink\simdemos\simfeatures\` verwenden. In diesen sowie in der SIMULINK-Hilfe finden Sie weitere Erläuterungen.

Aufgaben und Übungen

4.18 ☞ *Doppelintegrierer als S-Function:* Öffnen Sie `sfuncont.m` (Befehl `open sfuncont`), speichern Sie die Datei als `doppint.m` in Ihrem Arbeitsverzeichnis, und simulieren Sie die Übergangsfunktion mit Hilfe des Blocks `s-function`. Ändern Sie anschließend die Funktion in einen Doppelintegrierer ($G(s) = \frac{1}{s^2}$).

Ändern Sie die Funktion von doppint, sodass ein Übertragungsfaktor als zusätzlicher Parameter angegeben wird. Maskieren Sie den Block in SIMULINK, und geben Sie den Parameter als Variable mit dem Erklärungstext „Übertragungsfaktor“ an.

4.19 ☞ *Das Auto als S-Function:* Öffnen Sie das Modell aus Aufgabe 4.3. Programmieren Sie dazu eine S-Function zur parallelen Simulation des Autos. Geben Sie F_{Roll}, M und C als Parametern der S-Funktion ein. Vergleichen Sie mit Hilfe von Scopes die Verläufe beider Modelle.

Tipp: Sie brauchen die Zustandsraumdarstellung des Autos.

$$\begin{cases} X = \begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} X(1) \\ X(2) \end{bmatrix} \Rightarrow \dot{X} = \begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ a \end{bmatrix} \\ \\ \dot{X} = \begin{bmatrix} v \\ {}^1\!/_M \left(F_{Trieb} - F_{Roll} - C \cdot v^2 \right) \end{bmatrix} \end{cases}$$

4.20 ☞ *PT1-Glied als S-Function:* Programmieren Sie eine S-Function zur Simulation eines PT1-Elements. Die Zustandsdarstellung des PT1 lautet:

$$\begin{aligned} \dot{x}(t) &= -\frac{1}{T} \cdot x(t) + \frac{k}{T} \cdot u(t) \\ y(t) &= x(t) \end{aligned}$$

4.21 ☞ *Inverses Einzelpendel als S-Function:* In dieser Aufgabe soll das dynamische Verhalten des inversen Einzelpendels (siehe Abb. 4.11) als S-Function modelliert werden. Mit Hilfe von S-Functions können Systeme sehr einfach anhand der zugehörigen linearen bzw. nicht-linearen DGL-Systeme in Zustandsraumdarstellung beschrieben werden. Die allgemeine Form eines nicht-linearen DGL-Systems in Zustandsraumdarstellung stellt sich wie folgt dar:

$$\dot{\mathbf{x}} = f_1(\mathbf{x}, \mathbf{u})$$

$$\mathbf{y} = f_2(\mathbf{x}, \mathbf{u})$$

wobei $\mathbf{x}$ den Zustandsgrößenvektor, $\mathbf{u}$ den Eingangsgrößenvektor und $\mathbf{y}$ den Ausgangsgrößenvektor darstellt.

Das inverse Einzelpendel lässt sich durch die translatorische Bewegung des Schlittens und die rotatorische Bewegung des Pendels

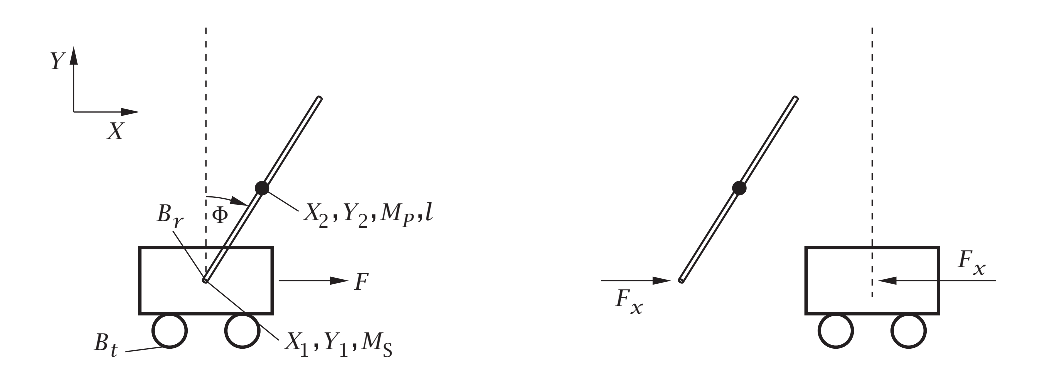


Abbildung 4.11: Inverses Einzelpendel

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & M_s + M_p & 0 & \frac{M_p \cdot l}{2} cos(\Phi) \\ 0 & 0 & 1 & 0 \\ 0 & cos(\Phi) & 0 & \frac{2}{3} l \end{pmatrix} \begin{pmatrix} \dot{X}_1 \\ \ddot{X}_1 \\ \dot{\Phi} \\ \ddot{\Phi} \end{pmatrix} = \begin{pmatrix} \dot{X}_1 \\ \frac{M_p \cdot l}{2} sin(\Phi) \cdot \dot{\Phi}^2 - B_t \cdot \dot{X}_1 \\ \dot{\Phi} \\ g \cdot sin(\Phi) - \frac{2 * B_r}{M_P \cdot l} \cdot \dot{\Phi} \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \cdot F$$

und die zugehörigen Parameter

Parameter	Beschreibung	Wert	Einheit
M_s	Masse des Schlittens	5	kg
M_p	Masse des Pendels	2	kg
l	Länge des Pendels	1	m
B_r	Reibungskoeffizient (Schlitten)	5	Ns/m
B_t	Reibungskoeffizient (Lagerreibung Pendel)	5	$N \cdot rad/s$

beschreiben.

Als Ausgangsgrößen werden die X-Position X_1 des Schlittens und der Winkel Φ verwendet:

$$\mathbf{y} = \begin{pmatrix} X_1 \\ \Phi \end{pmatrix}$$

Implementieren Sie die Bewegungsgleichungen des Systems als S-Function. Verwenden Sie dabei folgende Zustandsgrößen:

$$\mathbf{x} = \begin{bmatrix} X_1 & \dot{X}_1 & \Phi & \dot{\Phi} \end{bmatrix} \text{ mit dem Anfangszustand } \mathbf{x_0} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

Tipp: Das nichtlineare DGL-System

$$\mathbf{Q} \cdot \dot{\mathbf{x}} = f^*(\mathbf{x}, \mathbf{u})$$

kann wie folgt in Zustandsraumdarstellung transformiert werden:

$$\dot{\mathbf{x}} = \mathbf{Q^{-1}} \cdot f^*(\mathbf{x}, \mathbf{u})$$

Die Inverse der **Q**-Matrix kann dazu in jedem Iterationsschritt berechnet werden.

4.7 Andere wichtige Blöcke und Features von Simulink

LookUp Tables

Mit `LookUp Tables` können Beziehungen zwischen Ein- und Ausgabedaten erzeugt werden, auch wenn diese nicht durch eine analytische Funktion beschrieben werden (oder diese nicht bekannt ist). Dies geschieht durch einige, diskrete Wertepaare, die man beispielsweise durch Messungen erhalten hat. Zwischenwerte werden von SIMULINK interpoliert. Die Lookup Tables können beispielsweise zum Abbilden des Verhaltens eines Motors eingesetzt werden, wenn nur einige Werte des Motorkennfelds verfügbar sind.

Aufgaben und Übungen

4.22 ☞ *LookUp Table in 2D:* Laden Sie die Datei `motorkennfeld.mat`, die Ihnen Ihr Betreuer zur Verfügung stellen wird. Das Motorkennfeld ist auf Abbildung 4.12 dargestellt. Stellen Sie den

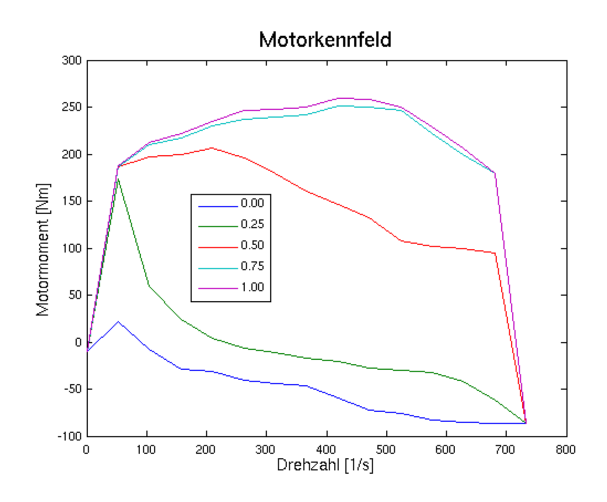
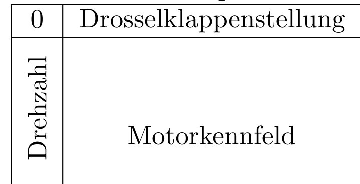


Abbildung 4.12: Motorkennfeld, Aufgabe 4.22

`Block Lookup Table (2-D)` so ein, dass er das statische Verhalten des Motors darstellt. Die nicht in der Datei vorhandenen Arbeitspunkte werden somit interpoliert. Die Matrix in der Datei hat folgende Form:



4.23 ☞ *Auto mit Steigungswiderstand:* Erweitern Sie das Modell aus Aufgabe 4.3 um den Steigungswiderstand. Die Steigung der Straße wird als eine Funktion der Position $x(t)$ gegeben und ist auf Abbildung 4.13 dargestellt. Nehmen Sie die Gleiche Parametern wie in Aufgabe 4.3, aber verwenden Sie ein Step $F_{Antrieb} = 5000N$ und eine Simulationszeit von $250sec$. Geben Sie $a(t)$, $v(t)$ und $x(t)$ in einem Scope mit 3 Achsen aus. Der Steigungswiderstand ist durch $F_{Steigung} = M \cdot g \cdot \sin(\alpha)$ beschrieben, wo α die Steigung in rad ist. Verwenden Sie den Block Lookup Table, um die Steigung über der Position im Modell zu beschreiben.

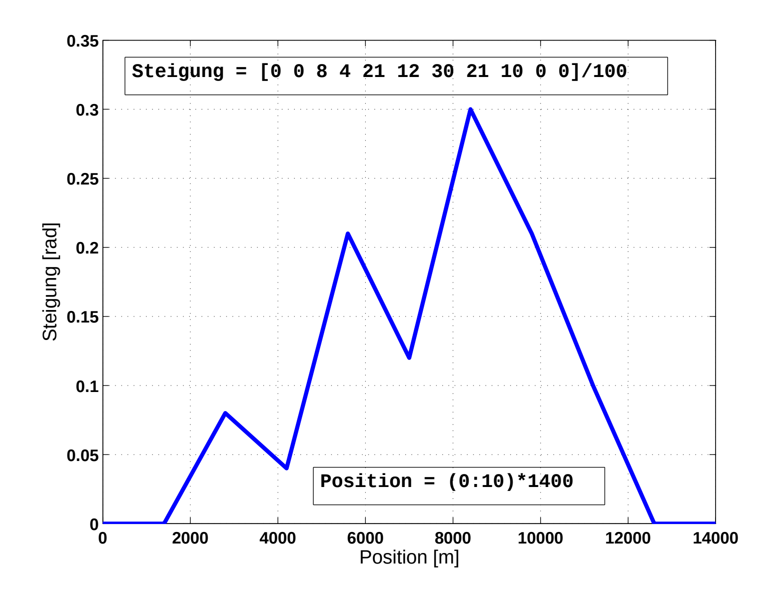


Abbildung 4.13: Steigungsprofil, Aufgabe 4.23

Erstellen einer eigenen Bibliothek

Simulink bietet die Möglichkeit, eigene Bibliotheken mit z. B. häufig benutzten Blöcken zu erstellen. Dazu wird ein neues Fenster über das Menü **File** → **New** → **Library** als Bibliothek und nicht wie sonst als Modell erstellt.

Alle in dieses Fenster kopierten Blöcke sind nach dem Abspeichern in Form einer neuen Bibliothek, die den Namen der Datei erhält, enthalten. Die Bibliothek kann später beliebig verändert und erweitert werden. Nach dem Abspeichern und Schließen ist die Bibliothek automatisch gesperrt und muss für gewünschte Veränderungen erst durch den Menü-Befehl **Edit** → **Unlock Library** freigegeben werden. Werden in Bibliotheken Änderungen an Blöcken vorgenommen, aktualisieren sich automatisch alle Modelle, in denen diese Blöcke vorkommen.

Durch Kopieren aus dem Bibliotheksfenster in ein Modellfenster werden Kopien der Bibliotheks-Blöcke erstellt. Die Verbindung wird dabei über den Blocknamen hergestellt, sodass dieser in der Bibliothek nicht mehr verändert werden sollte.

Aufgaben und Übungen

4.24 ☞ Erzeugen Sie eine Bibliothek mit dem PT1-Element als Inhalt.

4.8 Diskrete Subsysteme in kontinuierlichen Systemen

Um kontinuierliche Systeme in einem Digitalrechner zu berechnen zu können, ist zunächst eine **Abtastung** und eine anschließende **Digitalisierung** nötig. Gerade in der Regelungstechnik wird mehr und mehr zu dieser Vorgehensweise übergegangen, da dies eine ganze Reihe von Vorteilen bietet:

- Automatisierungsysteme sind naturgemäß binäre und digitale Verarbeitungsysteme
- Digitale Regler und Algorithmen sind flexibler als analoge PID-Regler
- Digitale Regler sind driftfrei und unempfindlicher gegenüber Umwelteinflüssen als Analogregler
- Gekoppelte Mehrgrößenregelungen sind häufig nur mit Digitalrechnern realisierbar (Zustandsregelung)
- Digitale Systeme sind meistens preiswerter als hochwertige analoge Systeme

Erfolgt die Abtastung eines kontinuierlichen System in so kurzen Zeitabständen, dass die sich daraus ergebene Wertfolge den kontinuierlichen Zeitverlauf nahezu genau wiedergibt, verhält sich die digitale Regelung in erster Näherung genauso wie eine analoge Realisierung. Man kann den Abtastvorgang also ignorieren. In der Praxis wird dies aber häufig nicht möglich sein, da Abtastvorgänge aufgrund der Verarbeitungsgeschwindigkeit des Automatisierugssystems nicht beliebig schnell durchgeführt werden können.

Mathematische Beschreibung von Abtastvorgängen

Auf eine ideale Abtastung erfolgt unmittelbar die Speicherung des Funktionswertes über die Abtastperiode T. Das gesamte Übertragungsglied, das aus der zeitkontinuierlichen Funktion $f(t)$ die Treppenfunkion $\bar{f(t)}$ erzeugt, wird als **Abtast-Halte-Glied** bezeichnet (Abbildung 4.14).

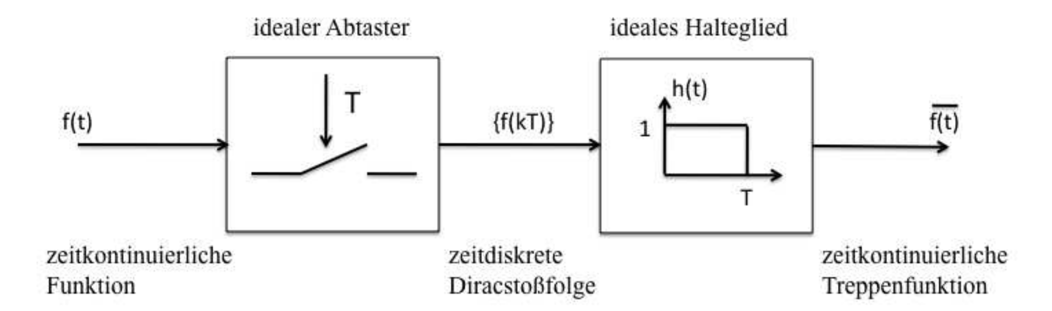


Abbildung 4.14: Ideales Abtast-Halte-Glied

Die Treppenfunktion $\bar{f(t)}$ entsteht also, indem die Funktion $f(t)$ zu jedem Zeitpunkt kT (für $k \geq 0$) abgetastet, und der entsprechende Funktionswert $f(kT)$ über die Abtastperiode T gehalten wird.

Um die Treppenfunkton $\bar{f(t)}$ mathematisch zu beschreiben, benötigt man den sogenannten **Rechteckimpuls** (Abbildung 4.15) in seiner auf die Abtastzeit T normierten Darstellung:

$$rect(\frac{t}{T}) = \begin{cases} \frac{1}{T} & |t| < \frac{T}{2} \\ \frac{1}{2} \cdot \frac{1}{T} \quad \text{für} & |t| = \frac{T}{2} \\ 0 & |t| > \frac{T}{2} \end{cases}$$

Die Normierung ist so gewählt, dass Rechteckimpuls die Fläche 1 hat. Es gilt also:

$$\int_{-\infty}^{+\infty} rect(\frac{t}{T})dt = 1$$

Multipliziert man den Rechteckimpuls mit T und verschiebt ihn um $\frac{T}{2}$ nach rechts erhält man die

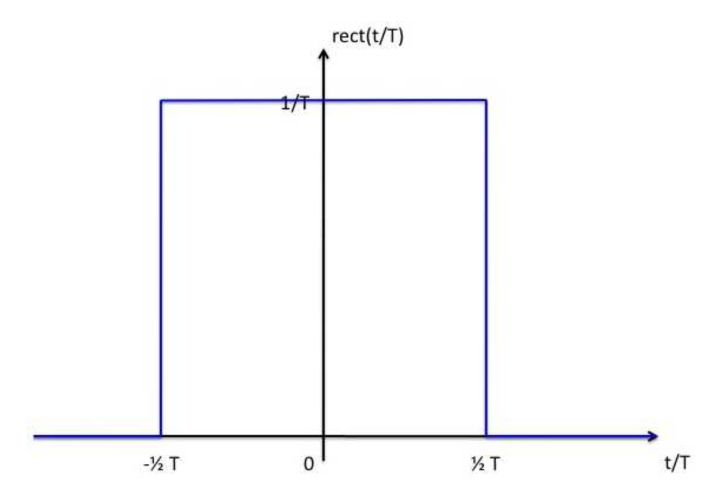


Abbildung 4.15: Rechteckimpuls $rect(\frac{t}{T})$

Haltefunktion 0. Ordnung (Abbildung 4.16):

$$h(t) = T \cdot rect(\frac{t - \frac{T}{2}}{T})$$

Die Treppenfunktion $f\bar{(}t)$ (Abbildung 4.17) berechnet sich nun mit Hilfe der diskreten Faltung

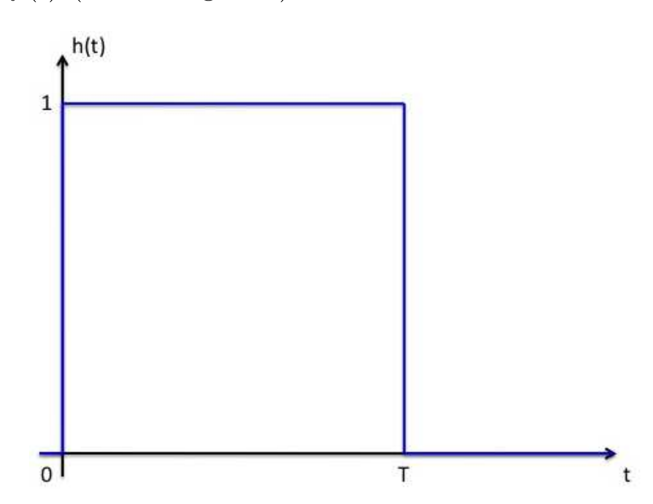


Abbildung 4.16: Haltefunktion 0. Ordnung $h(t)$

der Funktion $f(t)$ und der Haltefunkton 0. Ordnung $h(t)$:

$$f(t) \approx f\bar{(}t) = \sum_{k=0}^{\infty} f(kT) \cdot h(t - kT)$$

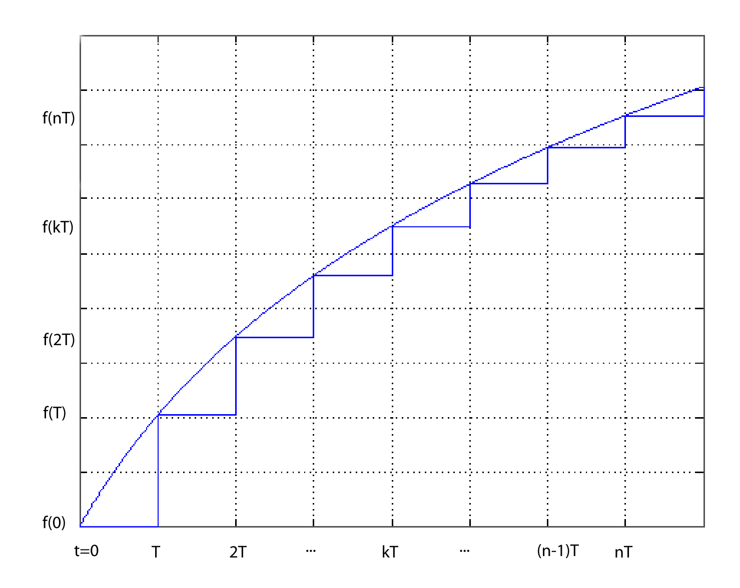
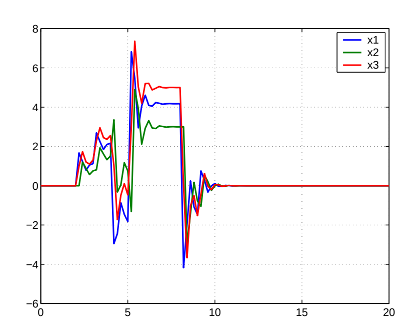


Abbildung 4.17: Treppenfunktion $f\bar{(t)}$

Aufgaben und Übungen

4.25 ☞ *Zeitdiskretes Zustandsraummodell:* Erstellen Sie ein zeitdiskretes Zustandsraummodell mit den selben Matrizen und Eingangs- signalen wie in der vorigen Aufgabe. Simulieren sie mit fester Integrationsschrittweite von 0.1 und diskretem Integrationsalgorithmus. Simulieren Sie anschließend mit der Integrationsschrittweite 0.02 und 2. Vergleichen Sie die Ergebnisse. Es soll sich folgender *Verlauf* der Zustände ergeben:



4.26 ☞ *Populationswachstum:* Die logistische Gleichung wurde ursprünglich 1837 von Pierre François Verhulst als demografisches Modell eingeführt. Der Grundgedanke dahinter ist folgender: Geht man davon aus, dass für ein gegebenes Jahr n die normalisierte Population x_n ist. Dann kann die Population des nächsten Jahres x_{n+1} näherungsweise als proportional zu der aktuellen Population x_n und zum verbleibenden bewohnbaren Raum $1 - x_n$ angenommen werden. Berücksichtigt man einen Parameter, der z.B. von der Fruchtbarkeit, von der bewohnbaren Anfangsfläche und von der gemittelten Sterberate abhängt, so kann man das Verhalten des Systems mit folgender Gleichung darstellen:

$$x_{n+1} = \lambda \cdot x_n \cdot (1 - x_n)$$

Erstellen Sie ein Modell im Simulink, das das demografische Wachstum simuliert und die Po-

pulation in einem Scope plotten kann. Es soll anhand dieses Modells die zukünftige Population vorhergesagt werden. Wird das Wachstum senken oder zyklisch variieren? Um diese Fragen zu beantworten Simulieren Sie das System für folgende Fälle und füllen Sie die untere Tabelle aus:

x_0	λ	Verhalten
$0,5$	$0,5$	
$0,5$	$1,5$	
$0,5$	$2,5$	
$0,5$	$3,2$	
$\frac{5}{16}$	$3,2$	
$0,5$	$3,55$	
$0,5$	$3,7$	
$0,5$	$4,2$	

Um die Tabelle auszufüllen, verwenden Sie folgende Begriffe (auch kombiniert): *stabil*, *instabil*, *mit Grenzwert = X*, *wellenförmig*, *monoton*, *mit einer/mehreren Periode/n*, *chaotisch* (die Werte wiederholen sich nicht).

Hinweis: Das chaotisches Verhalten ist hier möglich, weil das System nicht linear ist.

5 Symbolisches Rechnen

Die Symbolic Math Toolbox enthält Werkzeuge zur Lösung und Manipulation symbolischer Ausdrücke und zur Durchführung von Berechnungen mit variabler Genauigkeit. Die Toolbox umfasst mehrere hundert symbolische MATLAB-Funktionen, die die MuPAD-Engine zur Differentiation, Integration, Vereinfachung, Umwandlung oder Lösung mathematischer Gleichungen nutzen.

Ein weiterer zentraler Bestandteil der Symbolic Math Toolbox ist die MuPAD-Sprache, die auf die Verarbeitung von symbolischen Ausdrücken und Operationen optimiert ist. Die Toolbox enthält Bibliotheken mit MuPAD-Funktionen für allgemeine Teilbereiche der Mathematik wie die Differenzial- und Integralrechnung oder die Lineare Algebra sowie für Spezialgebiete wie die Zahlentheorie und die Kombinatorik.

5.1 Symbolische Objekte

Symbolische Variablen

Möchte man eine einzelne Variable deklarieren, kann der Befehl `sym` wie folgt benutzt werden:

```
>> a = sym('a1')
```

Der Befehl `syms` ermöglicht das Deklarieren mehrerer Variablen, die sich wie in der Mathematik verarbeiten lassen.

```
>> syms x y z
>> x+x+y
```

Neben einzelnen Variablen lassen sich auch Arrays (Matrizen und Vektoren) deklarieren und können durch übliche Operationen verarbeitet werden.

```
>> syms a11 a12 a21 a22;
>> A  = [a11 a12;a21 a22]
>> A^2
```

Standardmäßig werden die symbolische Variablen als *komplex* deklariert. Die Erzeugung *realer* oder *positiver* Variablen kann wie folgt gemacht werden:

```
>> syms x real;
>> syms y positive;
```

Symbolische Zahlen

Mit der symbolischen Toolbox können Zahlen in symbolische Objekte konvertiert werden. Für diese Konvertierung wird auch der Befehl `sym` benutzt.

```
>> a=sym('2')
>> sqrt(a)
```

Der numerische Wert einer symbolischen Zahl lässt sich dann mit dem Befehl `double(a)` gewinnen. Dank solcher symbolischen Zahlen kann man mit Bruchzahlen exakte Ergebnisse erhalten:

```
>> sym(2)/sym(5)  % Effizienter mit sym(2/5)
```

5.2 Symbolische Funktionen

Nicht nur symbolische Variable, sondern auch symbolische Funktionen lassen sich mit der Toolbox erstellen.

```
>> syms a b c  % Deklaration symbolischer Variablen
>> f = a^2 + b + c;
```

Diese Funktionen können nachher mit üblichen mathematischen Operationen manipuliert werden. Darüber hinaus ist es möglich, Gleichungen aufzulösen und Variablen nummerische Werte oder mathematische Ausdrücke zuzuweisen.

Substitution von symbolischen Variablen

Variablen in einer symbolischen Funktion können anhand des Befehls `subs` durch Werte ersetzt werden. Die Syntax für das obige Beispiel sieht wie folgt aus:

```
>> subs(f, a, 3)   % Es wird a=3 eingesetzt
>> subs(f,b,c^2)   % Es wird b=c^2 eingesetzt
```

Das erste Argument ist der Funktionsname, das Zweite ist der Name der symbolischen Variable, die ersetzt werden soll und das Dritte ist die neue Variable, Ausdruck oder zugewiesener Wert.

```
>> subs(f, a, 'x')    % '' sind notwendig, wenn x nicht deklariert wurde
```

Eine mehrfachige Zuweisung ist hier möglich. Die Syntax sieht folgendermaßen aus:

```
>> subs(f, [a,b], [1,2*c])
```

Werden alle symbolische Variablen der Funktion durch einen Wert ersetzt, so wird das Ergebnis automatisch als double gegeben.

```
>> r=subs(f, [a,b,c], [1,4,3])
>> class(r) % Was für einen Wert ist r (double)
```

Gleichungen und Gleichungssysteme

Die symbolische Toolbox enthält eine Menge vordefinierter Funktionen zur Verarbeitung von symbolischen Ausdrücken. Mit Hilfe der Funktion `factor` kann beispielsweise eine Funktion faktorisiert werden. Der Befehl `expand` ermöglicht das Ausmultiplizieren und `collect` das Gruppieren von beliebigen Variablen.

```
>> syms a b x
>> f = a^2 - 2*a*b + b^2;
>> f1 = factor(f)              % Faktorisieren
>> expand(f1)                  % Wieder Ausmultiplizieren
>> collect(x^2+a*x+b+2*x,x)    % Gruppieren nach x
```

Mathematische Operationen mit symbolischen Funktionen führen meistens zu komplizierten Ausdrücken. Der Befehl `simplify` bringt den Ausdruck in seine vereinfachte Form.

```
>> f2=(6*x + 6)/(x^2 + x - 3) - ((2*x + 1)*(3*x^2 + 6*x - 1))/(x^2 + x - 3)^2
>> simplify(f2)
```

5.3 Algebraische Gleichungen

Mit der symbolischen Toolbox können algebraische Ausdrücke, Gleichungen und Gleichungssysteme gelöst werden.

Lösung einer algebraischen Gleichung

Eine Gleichung kann nach einer gewählten Variable aufgelöst werden, wobei die restliche als Parameter aufgefasst werden. Der dafür notwendige Befehl ist `solve`, dessen Verhalten durch zusätzliche Argumenten beeinflusst werden kann (siehe die Hilfe).

```
>> syms x y
>> f = 5*x^2 - 2*x^2*y + x*y^2 - x*y + y^3 - 4*y^2;  % Gleichung

>> solve(f,y)               % Auflösung der Gleichung f=0 nach y
```

Man kann sehen, dass das Ergebnis nur von `x` abhängt. Wird keine Variable gewählt (`solve(f)`), dann wird nach der Variable, die als erste in der Alphabet kommt (hier `x`), automatisch aufgelöst.

Gleichungssysteme

Gleichungssysteme werden auch mit Hilfe von `solve` gelöst. Soll z.B. ein lineares Gleichungssystem mit 2 Unbekannten (x, y) und Gleichungen $a\,x+y=3$ und $2\,x+2\,y=5$ mit a als Parameter gelöst werden, können folgende Befehle verwendet werden:

```
>> syms a x y
>> f(1) = a*x + y - 3;              % Gleichung 1 (=0)
>> f(2) = 2*x + 2*y - 5;            % Gleichung 2 (=0)
>> [X,Y] = solve(f(1), f(2),'x,y')  % Lösung nach x und y
```

Wenn das Ergebnis einer Variable `Ergebnis` zugewiesen wird, liefert Matlab das Ergebnis als Struktur zurück. Mit `Ergebnis.a` kann wie üblich der Wert von `a` zugegriffen werden.

5.4 Infinitesimalrechnung in Matlab

Für die Infinitesimalrechnung wird das MuPAD-Engine verwendet. MuPAD ist ein Computeralgebrasystem, das ursprünglich an der Universität Paderborn entwickelt wurde. Es wurde später von The MathWorks gekauft und in Matlab integriert.

Integration

Die Integration erfolgt mit dem `int`-Befehl. Als Beispiel sollte die Integration der Funktion $f = x^2 - 2x - 4$ berechnet werden.

```
>> syms x
>> f=x^2-2*x-4;
>> g=int(f)      % Unbestimmtes Integral
```

Die Integration der Funktion von a bis b erfolgt mit dem Befehl `int(f,a,b)`. Eine Variante ist der Befehl `int(f,v,a,b)`, wenn man nach `v` integrieren möchte. Sollte Matlab keine geschlossene Lösung des Integrals finden, erscheint das auf dem Bildschirm.

Ableitung

Die Ableitung erfolgt genauso einfach wir die Integration mit dem `diff`-Befehl. Zur Verifizierung des obigen Beispiels kann man g nach x ableiten.

```
>> diff(g,x)
```

Eine höhere Ableitung erfolgt mit dem Befehl `diff(g,n)`, wobei `n` die Ordnung der Differenzierung ist. Wenn der Ausdruck mehrere symbolische Variable enthält, kann man festlegen nach welcher Variable abgeleitet werden soll. Sei beispielsweise der symbolische Ausdruck `f=sin(s*t)` gegeben, dann kann mit dem Befehl `diff(f,t)` die partielle Ableitung von `f` nach `t` berechnet werden. Wenn keine Variable festlegt wurde, wird nach der Variable, die als erste in der Alphabet kommt, abgeleitet. Eine Ableitung höherer Ordnung erfolgt mit `diff(f,t,n)`.

Jacobian

Die Jacobi-Matrix einer differenzierbare Funktion oder eines Vektors von Funktionen lässt sich einfach mit der symbolischen Toolbox berechnen.

Sei als Beispiel ein Punkt in kartesischer Koordinaten (x, y, z) gegeben. Dieser kann auch mit Kugelkoordinaten (r, θ, ϕ) dargestellt werden. Die Koordinatentransformation ist durch

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r\,\cos(\theta)\cos(\phi) \\ r\,\sin(\theta)\sin(\phi) \\ r\,\cos(\theta) \end{bmatrix}$$

beschrieben. Die Jacobi-Matrix dieser Koordinatentransformation ist die Matrix der partiellen Ableitungen, die sich mit der Funktion `jacobian` der symbolischen Toolbox berechnet lässt.

```
>> syms r l f
x = r*cos(l)*cos(f); y = r*cos(l)*sin(f); z = r*sin(l);
J = jacobian([x; y; z], [r l f])
```

So können dann die Geschwindigkeiten in Kugelkoordinaten $(\dot{r}, \dot{\theta}, \dot{\phi})$ zur kartesischen Koordianten transformiert werden.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \dot{r} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix}$$

Damit die Trasnformation eindeutig ist, muss die Matrix **J** invertierbar ($det(\mathbf{J}) \neq 0$) sein. Die Determinante der Matrix liefert einen komplizierten Ausdruck, welcher mit dem Befehl `simple` vereinfacht werden kann.

```
>> DetJ=simple(det(J))
```

Die Jacobi-Matrix kann auch verwendet werden, um Ableitungen mit Hilfe von der Kettenregel symbolisch darzustellen. So kann man beispielsweise die Geschwindigkeit der Koordinate x in Abhängigkeit von der Geschwindigkeiten in Polarkoordinaten berechnen: $\dot{x} = \mathbf{J_x} \begin{bmatrix} \dot{r} & \dot{\theta} & \dot{\phi} \end{bmatrix}^T$

```
>> syms r l f vr vl vf      % Position- und Geschwindigkeitskoordinaten
>> vx = jacobian(r*cos(l)*cos(f),[r,l,f])*[vr;vl;vf]
```

5.5 Graphische Darstellungen

Die Symbolic Toolbox von Matlab besitzt ebenfalls eine Reihe von graphischen Möglichkeiten:

- Plotten von expliziten Funktionen
- Plotten von impliziten Funktionen
- Plotten von 3D-Kurven
- Plotten von Flächen

Plotten von expliziten und impliziten Funktionen

Der Befehl zum Plotten von Funktionen in 2D lautet `ezplot`. Die Beschriftung erfolgt genauso wie mit üblichen plots.

```
>> syms x
>> f=sin(x)
>> ezplot(f);
>> hold on;
>> xlabel('x axis');
>> ylabel('y axis');
>> title('Explicit function: sin(2*x));
>> grid on;
```

Möchte man jetzt die Funktion nur im Bereich $x \in (-2\pi, 2\pi)$ plotten.

```
>> f=sin
>> ezplot(f,[-2*pi 2*pi]);
```

Die implizite Funktionen können wie folgt geplottet werden.

```
>> syms x y
>> f = x^2+y^2-1 % Ein Kreis (f=0)
>> ezplot(f);
```

Darstellung von Kurven in 3D

Der Befehl `ezplot3` wird benutzt, um 3D-Kurven zu erstellen. Damit kann z.B. die Trajektorie eines Flugzeuges im 3D-Raum dargestellt werden.

```
>> syms t
>> ezplot3(t^2*sin(10*t), t^2*cos(10*t), t);
>> xlabel('xlabel('ezplot3(t^2*sin(10*t)');
>> ylabel('t^2*cos(10*t)');
```

Darstellung von Oberflächen

Man benutzt den `ezsurf` Befehl zur Darstellung von Oberflächen. Als Beispiel wird hier die Funktion `Z=X^2-Y^2` dargestellt.

```
>> syms x y
>> ezsurf(x^2 - y^2);
>> hold on;
>> zlabel('z');
>> title('z = x^2 - y^2');
```

5.6 Transformationen

Fourier-Transformation

Die Fourier-Transformation ist eine Integraltransformation, die einer gegebenen Funktion $f(t)$ eine andere Funktion $F(\omega)$ (ihre Fourier-Transformierte) zuordnet. Sie ist eng mit der Laplace-Transformation verbunden. In vielen Einsatzgebieten wird sie dazu verwendet, um für zeitliche Signale (z. B. ein Sprachsignal oder einen Spannungsverlauf) das Frequenzspektrum zu berechnen. Um die Fourier-Transformation einer Funktion in Matlab zu berechnen, wird der Befehl `fourier` benutzt. Als Beispiel soll die Fourier-Transformation der Funktion $f(t) = e^{-t^2}$ berechnet und im Intervall $(-5, 5)$ geplottet werden.

```
>> syms t
>> f=exp(-t^2); subplot(2,1,1); ezplot(f,[0 3]);
>> Fourierf=fourier(f)
>> subplot(2,1,2); ezplot(Fourierf,[-5 5])
```

Der Befehl `fourier(f)` (mit Standardparameter `t`) führt zu einer Transformation `F(w)` (mit Standardparameter `w`). Die Rücktransformation kann mit dem Befehl `ifourier` gewonnen werden.

```
>> iFourierf=ifourier(Fourierf)
```

Laplace-Transformation

Die Laplace-Transformation kann als Verallgemeinerung der Fourier-Transormation aufgefasst werden. Die Laplace-Transformation bildet die reelle Originalfunktion $f(t)$ in einer komplexen veränderlichen Funktion $F(s)$ ab. In Matlab wird der Befehl `laplace` benutzt, der analog zu `fourier` verwendet werden kann.

```
>> syms t
>> f=1-exp(-t);
>> Fs = laplace(f)
```

Die Rücktransformation kann mit dem Befehl `ilaplace` gewonnen werden.

Aufgaben und Übungen

5.1 ☞ *Lösung der quadratischen Gleichung:* Berechnen Sie die Wurzel der Gleichung $a\,x^2 + b\,x + c = 0$ und weisen Sie das Ergebnis der Variable `loseung` zu. Verwenden Sie `subs`, um die Lösung für $(a, b, c) = (1, 2, 3)$ zu berechnen.

5.2 ☞ *Symbolische Matrizen:* Berechnen Sie die Determinante, die Eigenwerte und die Inverse folgender Matrix:

$$\mathbf{A} = \begin{bmatrix} a & 2 \\ 3 & b \end{bmatrix}$$

Setzten Sie $b = 2\,a$ ein und finden Sie a so, dass die Matrix singulär ist.

5.3 ☞ *Gleichungssystem:* Berechnen Sie die Lösung folgendes Gleichungssystems:

$$\begin{cases} 2x + b\,y + 3z & = 2 \\ a\,x + y + 4z & = 7 \\ x + y + z & = 3 \end{cases}$$

wobei a, b und c Parameter sind.

5.4 ☞ *Vereinfachung von Ausdrücken:*

- Multiplizieren Sie $f_1 = (x-3)^3(y+2)^2$ aus und weisen Sie das Ergebnis der Variable `f2` zu.
- Faktorisieren Sie `f2`.
- Gruppieren Sie `f2` nach x.
- Verifizieren Sie, dass $f_4 = \sin(x)^2 + \cos(x)^2$ gleich 1 ist.

5.5 ☞ *Infinitesimalrechnung:* Berechnen Sie folgendes:

- Integrieren Sie $f(x) = x^2$ von a bis b.
- Leiten Sie das Ergebnis nach x ab.
- Integrieren Sie $f(x) = e^{-x}$ von a bis ∞. *Tipp:* `inf`.
- Eine Masse in der Ebene bewegt sich mit einer Geschwindigkeit $(\dot{r}, \dot{\phi}) = (1, 0.1)$ in Polarkoordinaten. Berechnen Sie die Position und Geschwindigkeit in kartesischen Koordinaten, wenn die aktuelle Position in Polarkoordinaten $(r, \phi) = (1, \pi/4)$ ist.

5.6 ☞ *Transformationen:* Berechnen Sie folgendes:

- Berechnen Sie die Laplace-Transformation von $f(t) = \sin(t)\, e^{-t}$
- Wenden Sie die inverse Transformation auf das Ergebnis an.
- Wie lautet die Fourier-Transformation von $\sin(t)\, t^2$?

5.7 ☞ *Graphische Darstellungen:* Bearbeiten Sie folgende Punkte:

- Plotten Sie die Funktion $f = \frac{\sin(x)}{x}$ mit `ezplot`.
- Plotten Sie $x^2 - y^2 = 3$.
- Plotten Sie die Kurve $(x, y, z) = (\sin(t), \cos(t), t)$.
- Plotten Sie die Oberfläche $z = x^2 + y^2$.

6 Control System Toolbox

MATLAB selbst ist eine Rechenumgebung, die die Entwicklung eigener Algorithmen ermöglicht. Bereits entwickelte Algorithmen werden nach Themengebieten geordnet in sogenannten **Toolboxen** angeboten. So existieren Toolboxen z. B. zur Signalverarbeitung, Identifikation, Reglerentwurf aber auch zur Statistik, Optimierung, ... Die Liste ist sehr umfangreich und wächst ständig.

Im Bereich der Regelungstechnik ist die Basis die **Control System Toolbox**, die hier auch vorgestellt werden soll. Weitere am Institut für Regelungstechnik vorhandene Toolboxen von Bedeutung sind (ohne Anspruch auf Vollständigkeit) die **Model Predictive Control Toolbox, Signal Processing Toolbox, System Identification Toolbox, Optimization Toolbox, LMI Control Toolbox, Robust Control Toolbox, Wavelet Toolbox** und **Neural Network Toolbox**.

6.1 LTI-Systeme in Matlab

Die Control System Toolbox kennt neue Datenstrukturen zur Darstellung linearer, zeitinvarianter Systeme (Linear Time Invariant: LTI-Systems). Sie werden über die drei parametrischen Funktionen `tf` (Übertragungsfunktion: Transfer Function), `zpk` (Pol-/Nullstellen-/Übertragungsfaktor: Zero/Pole/Gain)), `ss` (Zustandsraum: State Space) und die nichtparametrische Funktion `frd` (Frequenzgang: Frequency Response Data) erzeugt. Im folgenden werden allein zur besseren Übersicht die entsprechend definierten Variablen `sys`$*$ genannt.

Zeitkontinuierliche Systeme

Die *Übertragungsfunktion* in MATLAB benötigt als Argumente für `tf` die Angabe zweier Zeilenvektoren, die die Koeffizienten von Zähler- und Nennerpolynom in fallender Potenz von s enthalten. Für eine Übertragungsfunktion

$$G(s) = \frac{3s-2}{0.25s^2+0.1s+1}$$

folgt somit

```
>> z = [3  -2];
>> n = [0.25  0.1  1];
>> sys1 = tf(z,n)
```

Tipp: Sie können auch Übertragunsgfunktionen direkt eingeben, wenn Sie davor den Befehl `s=tf('s')` ausführen. Danach kann beispielsweise direkt `G=(s+1)/(s+2)` im Command Window getippt werden.

Bei der Darstellung von dynamischen Systemen durch die *Zustandsraumdarstellung* wird das Systemverhalten über gekoppelte Differentialgleichungen 1. Ordnung beschrieben,

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{Ax}+\mathbf{Bu}\\ \mathbf{y} &= \mathbf{Cx}+\mathbf{Du}\end{aligned}$$

und in Form der vier Matrizen **A**, **B**, **C** und **D** in MATLAB abgespeichert. Mit **x** wird der Vektor der Zustandsgrößen bezeichnet. Für MIMO (**M**ulti-**I**nput-**M**ulti-**O**utput)-Systeme können die Ein- und Ausgangsgrößen **u** und **y** auch in vektorieller Form auftreten.

Die Transformation der oben vorgestellten Übertragungsfunktion in die Zustandsraumdarstellung, d. h. die Berechnung und Darstellung der vier Matrizen **A**, **B**, **C** und **D** aus den Zeilenvektoren des Zähler- und Nennerpolynoms des Übertragungsfunktions-Modells `sys1`, ist sehr einfach: Der Befehl `ss` muss nur als Argument das vorher definierte Modell `sys1` erhalten:

```
>> sys2 = ss(sys1);
```

Die Eigenwerte der Matrix **A** kann man sich mit dem Befehl `eig(sys2.a)` ausgeben lassen. Um die Position der Polstellen (Eigenwerte) und Nullstellen auf der komplexen Ebene (PN-Bild) graphisch darzustellen, kann der Befehl `pzmap(sys2)` verwendet werden. Die Berechnung der Eigenfrequenzen und Dämpfungen eines Systems kann mit Hilfe des Befehls `damp(sys1)` erfolgen.

Der Vollständigkeit halber sei noch kurz die Syntax von `zpk` und `frd` aufgeführt. Für ausführliche Erläuterungen rufen Sie bitte die Hilfe auf.

```
>> z = 0.5; p = [-1, 2]; k = 3;  % Nullstelle bei 0.5, Polstellen bei -1 und 2,
                                 % Übertragungsfaktor 3

>> sys3 = zpk(z, p, k)
>> fgang= [3-i, 8-4i, 2-3i];
>> freq = [0.1, 1, 10];          % Werte des Frequenzgangs an den Frequenzen 0.1,
                                 % 1 und 10 rad/s
>> sys4 = frd ( fgang, freq);
```

Allgemein erfolgt eine Umwandlung zwischen den verschiedenen Darstellungsformen eines Modells automatisch, indem den Befehlen `tf, ss` oder `zpk` allein ein in anderer Darstellung definiertes parametrisches Modell (nicht frd!) als Argument gegeben wird. Einzige Ausnahme bildet die nicht-parametrische Darstellung in `frd`. Sie wird aus den anderen mit einer zusätzlichen Angabe über den Frequenzvektor erzeugt. Umgekehrt können aber die parametrischen Darstellungen nicht mehr aus `frd` gewonnen werden!

Die bei den vier Funktionen zur Erzeugung von Modellen angegebenen Argumente können aus den Modellen `sys*` wieder mit Hilfe der Funktionen `tfdata, ssdata, zpkdata` und `frdata` gewonnen werden. Näheres hierzu in der MATLAB-Hilfe. Die auf LTI-Strukturen gespeicherte Informationen können oft auch durch direkte Zuweisung veändert werden, so kann man beispielsweise den Zähler von `sys1` mit dem Befehl `sys1.num=[1 2]` anpassen. Um einen Überblick über die Felder einer LTI-Datenstruktur zu gewinnen, kann der Befehl `get(sys2)` angewendet werden.

Zeitdiskrete Systeme

Bisher wurde davon ausgegangen, dass es sich um die Darstellung zeitkontinuierlicher Systeme handelt. Analog zur Differentialgleichung im zeitkontinuierlichen existiert für den zeitdiskreten Bereich eine das dynamische Systemverhalten beschreibende Differenzengleichung:

$$\alpha_m y_{k-m} + \cdots + \alpha_1 y_{k-1} + \alpha_0 y_k = \beta_m u_{k-m} + \cdots + \beta_1 u_{k-1} + \beta_0 u_k$$

Die diskrete Übertragungsfunktionen lautet

$$G(z) = \frac{\beta_m z^{-m} + \cdots + \beta_1 z^{-1} + \beta_0}{\alpha_m z^{-m} + \cdots + \alpha_1 z^{-1} + \alpha_0} = \frac{\beta_m + \cdots + \beta_1 z^{m-1} + \beta_0 z^m}{\alpha_m + \cdots + \alpha_1 z^{m-1} + \alpha_0 z^m}$$

und die diskrete Zustandsraumbeschreibung:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{A}_d\mathbf{x}_k + \mathbf{B}_d\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k\end{aligned}$$

Anknüpfend an die kontinuierlichen Beispiele lassen sich zeitdiskrete Systeme durch das zusätzliche Argument einer Abtastzeit direkt angeben

```
>> Tabt = 0.2 % Abtastzeit Tabt = 0,2s
>> sys1dd = tf(z,n,Tabt)
>> sys2dd = ss(sys2.a,sys2.b,sys2.c,sys2.d,Tabt);
```

Mit Hilfe des Befehls `c2d` (continuous to discrete) lassen sich diskrete Modelle aus den kontinuierlichen Modellen gewinnen. Der Befehl `d2d` (discrete to discrete) ermöglicht eine spätere Änderung der Abtastzeit und der Befehl `d2c` (discrete to continuous) erzeugt wieder ein kontinuierliches Modell.

Hierfür ist sehr wichtig eine sinnvolle Abtastzeit zu nehmen, sodass die Signalen dann richtig abgetastet werden und die Information im Signal durch z.B. aliasing nicht verloren geht. Eine Regel aus der Praxis besagt, dass man die kleinste Zeitkonstante im System als Referenzpunkt nehmen soll und eine Abtastzeit $T_{abt} < T_{min}/(5,10)$ auswählt. Anhand des Befehls `pzmap` kann man überprüfen, dass die kleinste Zeitkonstante des Systems sys1 $1,5s$ beträgt. Dementsprechend wäre hier eine Abtastzeit von $0,2s$ sinnvoll.

```
>> sys1d = c2d(sys1, Tabt)
>> sys2d = c2d(sys2, Tabt);
>> Tabt2 = 2                  % Abtastzeit Tabt2 = 2s
>> sys1d2 = d2d(sys1d, Tabt2)
>> sys1c = d2c(sys1d)         % Vergleiche mit sys1
```

Warum unterscheiden sich die beiden zeitdiskreten Modelle 'sys1dd' und 'sys1d' sowie 'sys2dd' und 'sys2d'?

Totzeitbehaftete Systeme

Totzeitsysteme können bei einem Modell *sys* mit Hilfe der Eigenschaft `'iodelay'` und anschließender Angabe der Dauer der Totzeit eingeführt oder verändert werden. Wenn das Modell `sys` bereits existiert, kann diese Eigenschaft mit dem Befehl `set` verändert werden. Existiert es noch nicht, wird die Eigenschaft bei der Erzeugung mit angegeben. Im zeitdiskreten Fall gibt die Totzeit die Anzahl der Abtastschritte an, sie muss also ganzzahlig sein.

```
>> sys1tot = sys1;
>> set(sys1,'ioDelay', 0.5); sys1tot       % Totzeit von 0.5s in sys1, oder
>> sys1.ioDelay = .5;                      % direkte Zuweisung

>> sys2dtot = sys2d;                       % Totzeit von 4 Abtastschritten
>> sys2dtot.ioDelay=4; sys2dtot            % in sys2d
```

```
>> systot = zpk(0.5, [-1, -2], 2.5, 'ioDelay', 0.75);   % Direkte Eingabe
>> systotd = zpk(0.5, [-1, -2], 2.5, 4, 'ioDelay', 2);
```

Zusammenfassung der Modelltransformationen

Die bisher in diesem Kapitel gezeigten Darstellungsformen dynamischer Systeme sind zum besseren Verständnis teilweise in der folgenden Graphik dargestellt. Die Funktionen und Darstellungsformen aus MATLAB sind grau hinterlegt.

Dynamische Systeme		**kontinuierlich**		**diskret**	
Übertragungsfunktion	$G(s) :=$	$tf(\mathbf{z}, \mathbf{n})$		$tf(\mathbf{z_d}, \mathbf{n_d}, \mathbf{T_{abt}})$	$:= G(z)$
(transfer function)			**c2d**		
		ss $\Downarrow \Uparrow$ **tf**	$\rightleftharpoons$	**ss** $\Downarrow \Uparrow$ **tf**	
			d2c		
Zustandsraum		$ss(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$		$ss(\mathbf{Ad}, \mathbf{Bd}, \mathbf{C}, \mathbf{D}, \mathbf{T_{abt}})$	
(state space)		$\Updownarrow$		$\Updownarrow$	
		gekop.Dgln 1. O.		gekop.Δgln 1. O.	

mit den Abkürzungen:	*Dgl:*	*Differentialgleichung*
	Δgl:	*Differenzengleichung*
	gekop.:	*gekoppelt*

6.2 Kopplung von Systemen

Es kommt oft vor, dass man mehrere LTI-Systeme miteinander koppeln möchte. Dies geschieht im Allgemeinen auf drei Arten. Bei der *parallelen Verknüpfung* werden zwei Systeme nebeneinander geführt und ihre Ausgänge mithilfe einer Addition zusammengeführt. Im Falle einer *seriellen Verknüpfung* wird der Ausgang des ersten Systems als Eingang des zweiten Systems verwendet. Beim Prinzip der *Rückführung* wird der Ausgang des ersten Systems ebenfalls als Eingang für das zweite System verwendet, allerdings führt der Ausgang des zweiten Systems wieder zurück zum Eingang des ersten Systems und beeinflusst ihn damit. Diese drei Arten der Kopplung kann man in MATLAB mit den Befehlen

```
>> sys_ser = series(sys1,sys2)      % koppelt die Systeme sys1 und sys2 in Reihe
>> sys_ser = sys1*sys2              % auch mit * möglich

>> sys_par = parallel(sys1,sys2)    % koppelt die Systeme sys1 und sys2 parallel
>> sys_par = sys1+sys2              % auch mit + möglich

>> sys_feed = feedback(sys1,sys2)   % koppelt die Systeme sys1 und sys2 mit
                                    % einer negativen Rückführung
>> s_feed_p = feedback(sys1,sys2,+) % koppelt die Systeme sys1 und sys2 mit
                                    % einer positiven Rückführung
```

verwirklichen. Weitere Informationen zu den Arten der Kopplung und der Verwendung finden Sie auch in der MATLAB-Hilfe.

6.3 Graphische Darstellungen

Nachdem die Erzeugung und Umwandlung dynamischer Systeme in MATLAB behandelt worden sind, geht es nun um die graphische Darstellung und Charakterisierung der Systeme.

Zeitbereich

Für die Berechnung und Darstellung der Antwort des Systems im Zeitbereich stehen die Funktionen `impulse` (Gewichtsfunktion=Antwort auf einen Einheits-Impuls), `step` (Übergangsfunktion=Antwort auf einen Einheits-Sprung) und `lsim` (Antwort auf ein beliebiges Eingangssignal) zur Verfügung. Werden diese Befehle ohne linksseitige Argumente aufgerufen, so wird die berechnete Antwort nicht im Befehlsfenster ausgegeben sondern geplottet.

```
>> G=tf(1,[10 2 1]);
>> [h, th]=step(G);                      % Sprungantwort
>> [g, tg]=impulse(G);                   % Impulsantwort
>> figure(1); plot(th,h,tg,g); grid;

>> figure(2); step(G,c2d(G,1)); grid;    % kont. und diskrete Antworten

>> t=0:.1:80;                            % Zeitvektor
>> u=t.^2.*(t<20)/400+(t>=20)-(t>30);    % Beliebiger Eingang
>> figure(3); lsim(G,u,t); grid;         % Entsprechende Systemantwort
```

Tipp: Die statische Verstärkung k_{stat} kann man *für stabile Systeme* mit der Funktion `dcgain` berechnen lassen.

Frequenzgang

Für die Darstellung des Frequenzgangs eines Systems gibt es ebenfalls mehrere Möglichkeiten. Die bekannteste sind das Bode-Diagramm `bode` und die Ortskurvendarstellung `nyquist`, auch Nyquist-Diagramm genannt.

```
>> figure(4);                       % Bode-Diagramme von drei Systemen
>> bode(G,c2d(G,1),c2d(G,10));      % Man sieht, dass das dritte System
>> grid;                            % nicht richtig abgetastet wird

>> figure(5); nyquist(G);           % Ortskurve
```

Weitere Möglichkeiten sind die Befehle `nichols`, `sigma` und `freqresp` (siehe die MATLAB-Hilfe für weitere Erläuterungen).

Wurzelortskurven

Das Wurzelortskurven-Verfahren ist ein halbgraphisches Verfahren zur Bestimmung der Polstellen der Übertragungsfunktion des geschlossenen Regelkreises. Im Gegensatz zu den behandelten Frequenzgangverfahren können mit dem Wurzelortskurven-Verfahren in der hier gewählten Darstellung nur solche Systeme untersucht werden, deren Übertragungsfunktionen bzw. Frequenzgänge gebrochen rationale Funktionen sind. Regelkreise mit Totzeitgliedern sind daher zunächst ausgeschlossen.

Betrachtet wird der Regelkreis in Abbildung 6.1, wobei $G_S(s)$ und $G_R(s)$ die Übertragungsfunktionen der Strecke und des Reglers sind.

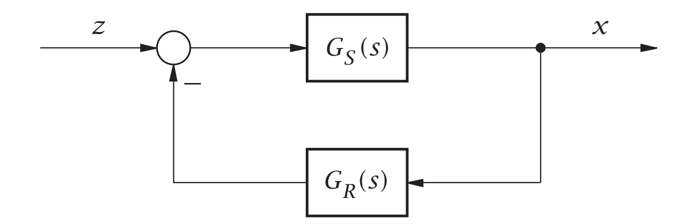


Abbildung 6.1: Standardregelkreis

Die Übertragungsfunktion des aufgeschnittenen Regelkreises ergibt sich dann als

$$G_0(s) = G_S(s) \cdot G_R(s) = K \frac{\prod_{i=1}^{m} (s - s_N i)}{\prod_{j=1}^{n} (s - s_P j)} = K \cdot G_0'(s) \tag{6.1}$$

und dementsprechend können das charakteristisches Polynom und Bedingungen für $G_0'(s)$ ermittelt werden:

$$1 + G_0(s) = 0 \Rightarrow K \cdot G_0'(s) = -1 \tag{6.2}$$

Die Wurzelortskurve (WOK) ist eine grafische Darstellung der Lage der Nullstellen des charakteristischen Polynoms in 6.2 (d.h. die Polstellen des geschlossenen Regelkreises) in Abhängigkeit des Parameters $0 < K < \infty$. Die Wurzelortskurve verdeutlicht somit die Verschiebung der Polstellen in Abhängigkeit des Parameters K und ermöglicht dadurch Rückschlüsse auf das Stabilitätsverhalten und die Dynamik des Regelkreises.

In MATLAB kann die Wurzelortskurve mit dem Befehl `rlocus` erzeugt werden. Sei z.B.

$$G_0(s) = Ki \cdot \frac{0,1s + 1}{s(s-1)},$$

so kann die Wurzelortskurve des Systems in Abhängigkeit vom Parameter Ki des Reglers dargestellt werden.

```
>> Go = tf([.1 1],[1 -1 0]);      % Ohne Parameter Ki!
>> rlocus(Go)
```

Das Ergebnis ist in Abbildung 6.2 dargestellt, wobei der Hintergrundraster eingeschaltet wurde. Außerdem wurde mit der Maus ein beliebiger Punkt der Kurve markiert, sodass die Eigenschaften des Systems für die gewählte Einstellung ($Ki = 251$) angezeigt werden.

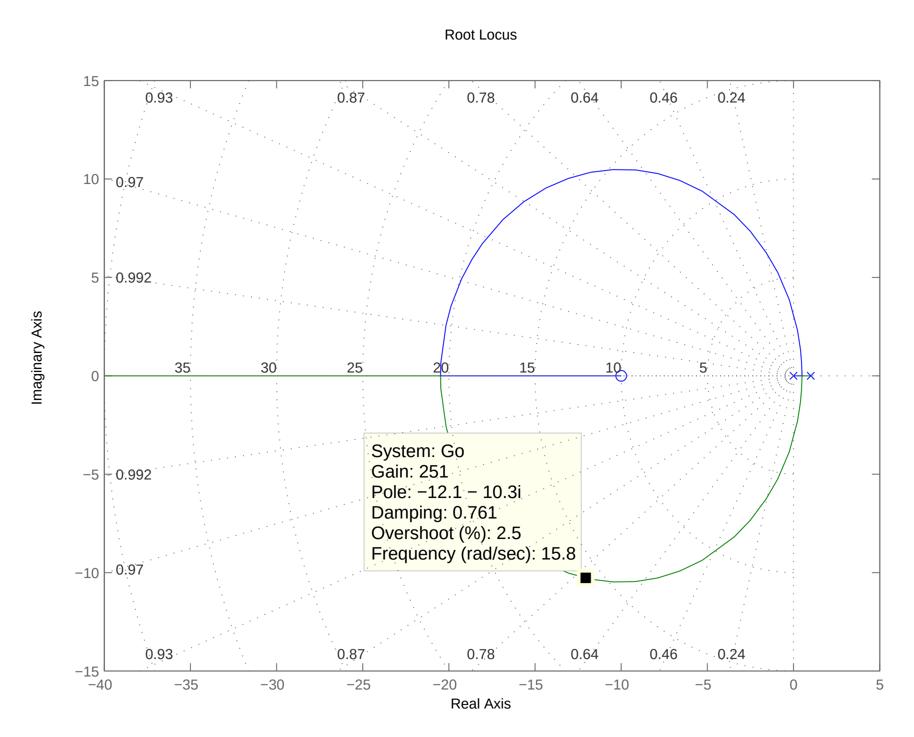


Abbildung 6.2: WOK von $G_0(s) = Ki \cdot \frac{0{,}1s+1}{s(s-1)}$

Das Wurzelortskurvenverfahren ist ein sehr wirksames Mittel, um die “Wanderung“ von Polstellen unter dem Einfluss von Änderungen bei Parametern eines Regelkreises darzustellen. Der Anwender sollte aber dabei nicht vergessen, dass die Dynamik eines Übertragungssystems nicht nur von den Polstellen sondern auch von den Nullstellen seiner Übertragungsfunktion abhängt.

6.4 Der LTI-Viewer, das SISO-Tool und Reglerentwurf

Eine sehr angenehme Möglichkeit zur übersichtlichen Darstellung und Analyse dynamischer Systeme bietet die graphische Oberfläche LTI Viewer (siehe Abb. 6.3), die mit dem Befehl `ltiview` aufgerufen wird. Sie ermöglicht die gleichzeitige Darstellung mehrerer Systeme sowie mehrerer Darstellungen nebeneinander und der einfachen Umschaltung zwischen verschiedenen Darstellungen. Diese Oberfläche ist größtenteils selbsterklärend, also probieren Sie sie einfach aus!

Eine Möglichkeit zur Analyse dynamischer Systeme, die nur einen Eingang sowie einen Ausgang haben (**S**ingle **I**nput **S**ingle **O**utput), bietet die ebenfalls in Matlab integrierte graphische Benutzeroberfläche `sisotool` (siehe Abb. 6.4). Hier können Kompensatoren sowie die Regelstrecke und der Regler eingestellt werden. Zusätzlich kann sowohl der offene als auch der geschlossene Regelkreis graphisch (z.B. in einem Bode-Diagramm) dargestellt werden. Man kann in `sisotool` jedes lineare Modell, dass man mit einer der oben genannten Methoden erstellt hat, einbinden.

Mit dem Befehl `lqr` (**Linear-Quadratisch-Optimale Zustandsrückführung (LQR)**) kann man die kostenminimierende Verstärkungsmatrix **K** durch Matlab bestimmen lassen. Die Kos-

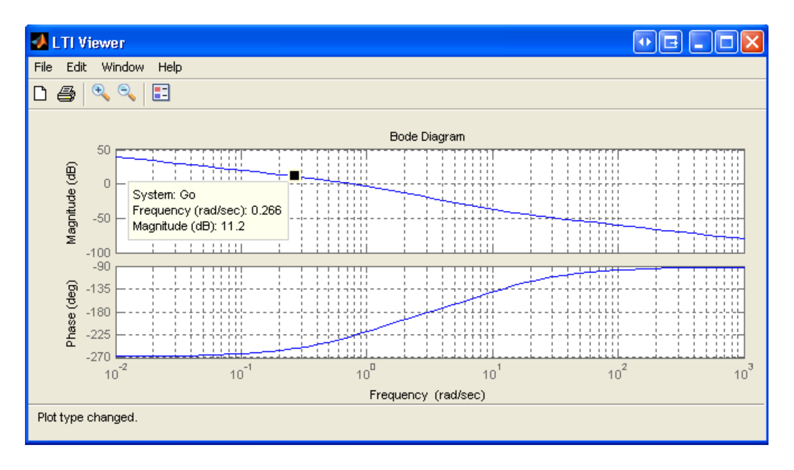


Abbildung 6.3: LTI-Viewer mit PT1-Glied

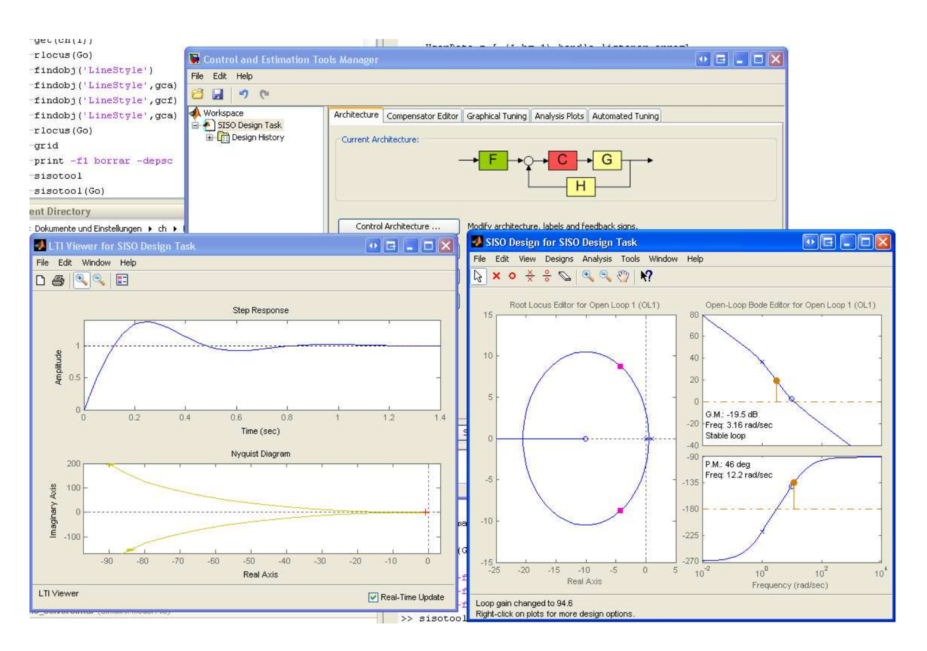


Abbildung 6.4: SISO-Tool

tenfunktion für kontinuierliche Systeme ist durch folgende Gleichung gegeben.

$$J = \int_0^\infty \left(x^T \mathbf{Q} x + u^T \mathbf{R} u + 2 x^T \mathbf{N} u \right) dt$$

Der Algorithmus liefert die Rückführungsmatrix $\mathbf{K}$, welche das optimale Regelgesetz $u = \mathbf{K}x$ definiert. Genauere Informationen hierzu findet man ebenfalls in der MATLAB-Hilfe. Eine weitere Möglichkeit bietet der Befehl `place` an, welcher die Rückführungmatrix $\mathbf{K}$ so berechnet, dass die Pole (Eigenwerte) des geschlossenen Regelkreises den Polen eines vorgegebenen Wunschpolynoms entsprechen (Polplatzierung).

Eine Übersicht aller Befehle der Control System Toolbox liefert der Befehl **help control**. Sehen Sie sich aber auch die Befehle **ltimodels** und **ltiprops** sowie aus der Signal Processing Toolbox **tf2ss** und **zp2ss** an.

6.5 Control System Toolbox in Simulink

In der Bibliothek *Control System Toolbox* von Simulink befindet sich der Block `LTI System`, der die Möglichkeit bietet, LTI-Modelle (`tf`, `ss`) in Simulink einzubinden. Darüber hinaus können unter *Tools:Control Design* Simulink-Modelle diskretisiert, analysiert und linearisiert werden. Hier wird nur kurz auf die Linearisierung von Simulink-Modellen eingegangen, weil diese nachher ermöglicht, die linearisierten Systeme in den Workspace zu exportieren und dort die bereits in den vorherigen Abschnitten behandelten Befehle zur Analyse und Entwurf anzuwenden.

Als Beispiel soll das nichtlineare System $\dot{x} = x^2 - x + u$ verwendet werden. Linearisert man diese Gleichung analytisch im Arbeitspunkt $x = 0$, so erhält man $\dot{x} = -x + u$. Die entsprechende Übertragungsfunktion ist $G(s) = \frac{1}{s+1}$, also ein PT1-Glied mit einer Zeitkonstante von $1s$.

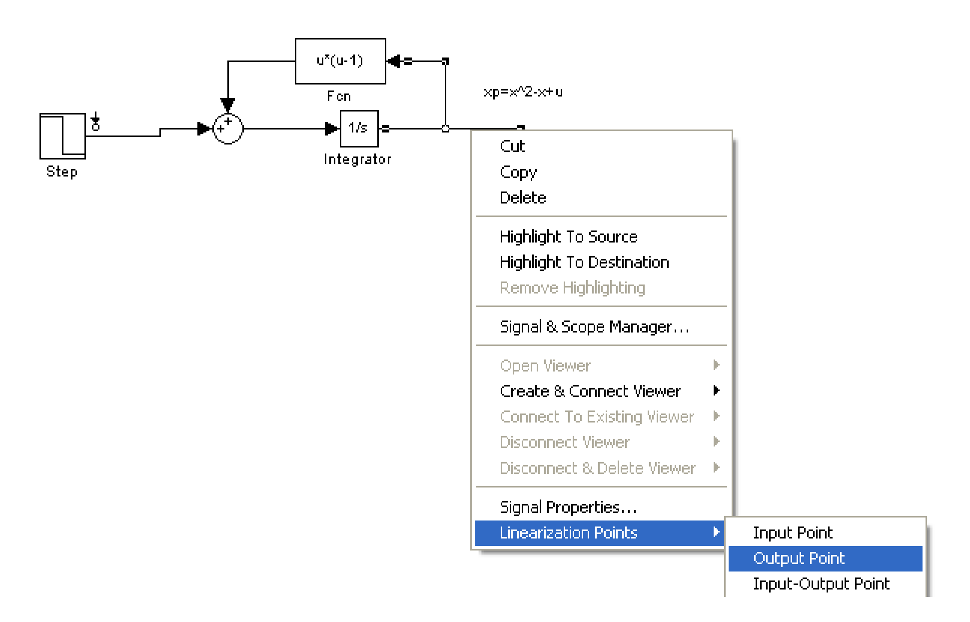


Abbildung 6.5: Linearisieren in Simulink

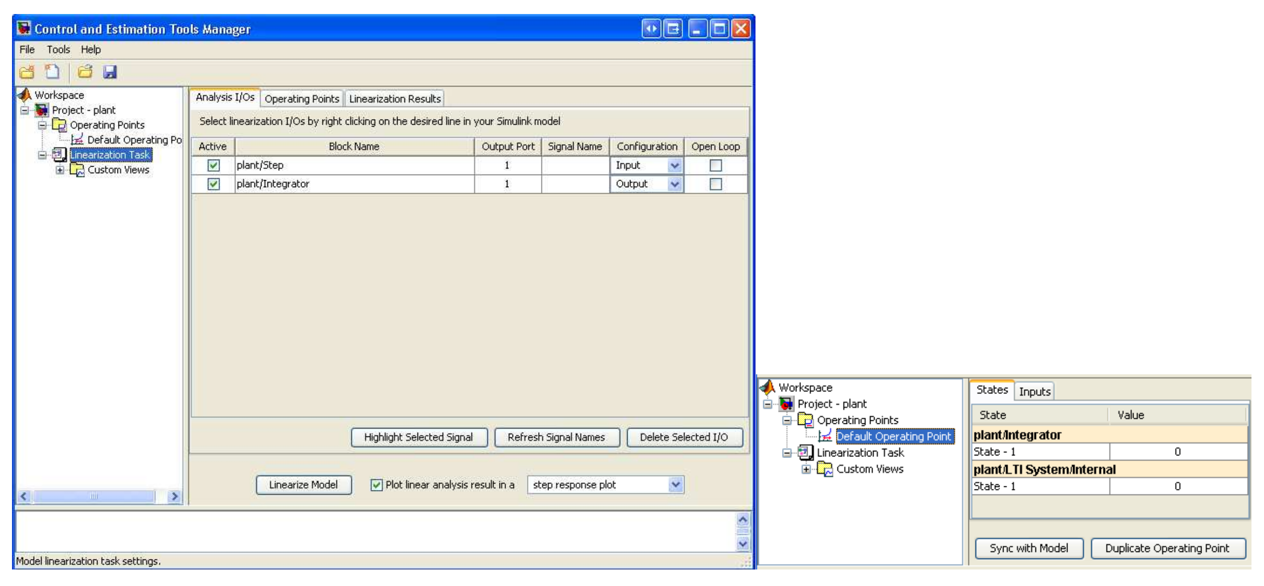
In Abb. 6.5 ist das entsprechende Simulink-Modell dargestellt. Der Integrator wird mit dem Wert 0 initialisiert. Um das Linearisierungsverfahren durchführen zu können, müssen erstmal die Ein- und Ausgänge definiert werden. Das erfolgt über das Kontextmenü *Linearization Points*, wie in der Abbildung gezeigt wird. Hier werden der Ausgang des Blocks `Step` als Eingang und der Zustand x als Ausgang ausgewählt.

Anschließend wird der **Control and Estimation Tools Manager**, der in der Abbildung 6.6 links dargestellt ist, über das Menü *Tools:Control Design:Linear Analysis* geöffnet.

Hier werden unter *Linearization Task* die Ein- und Ausgänge gezeigt, die in Simulink schon definiert wurden (in diesem Fall der Ausgang des Integrators und der Zustand x).

Unter **Operating Points** (siehe Abb. 6.6 rechts) können Arbeitspunkte berechnet und hinzugefügt werden. Hier beschränken wir uns auf nur einen Arbeitspunkt, der unter *Default Operating Point* eingestellt werden kann. Da das Modell nur einen Zustand besitzt und der Arbeitspunkt bei Null liegt, muss der Parameter `State - 1` gleich 0 sein.

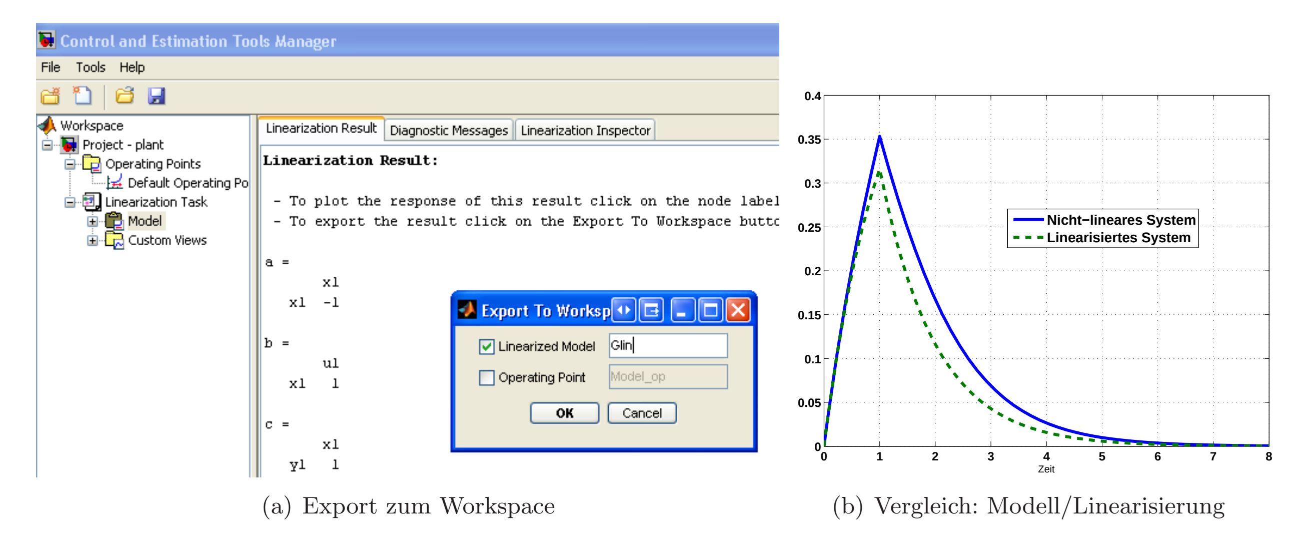
Um die Linearisierung durchzuführen muss der Knopf *Linearize Model* unter dem Tab *Linearization Results* betätigt werden. Das Ergebnis wird dann direkt mit dem LTI-Viewer dargestellt



(a) Ein- und Ausgänge (b) Arbeitspunktauswahl

Abbildung 6.6: Control Estimation Tools Manager

und die entsprechenden Gleichungen können unter *Linearization Task:Model* gefunden werden. Will man das dadurch gewonnenes LTI-Modell in den Workspace exportieren, dann kann man im Kontextmenü von *Model* die Option *Export* auswählen (siehe Abb. 6.7 links).



(a) Export zum Workspace (b) Vergleich: Modell/Linearisierung

Abbildung 6.7: Export des linearisierten Modells und Vergleich mit dem Nicht-linearen Modell

Jetzt können die Antworten beider Systeme verglichen werden, indem ein LTI-System-Block mit dem linearisierten Modell und demselben Eingang hinzufügt wird und beide Ausgaben in einem `Scope` dargestellt werden. Nimmt man als Eingangssignal einen Sprung $\begin{cases} u = 0,5 & \forall t < 1s \\ u = 0 & \forall t \geq 1s \end{cases}$, dann kann man sehen, dass sich der Unterschied bemerkbar macht, wenn sich das System vom Arbeitspunkt $x = 0$ entfernt (siehe Abb. 6.7 rechts).

Aufgaben und Übungen

6.1 ☞ *Erstellen von zeitkontinuierlichen LTI-Modellen:*

a. Erstellen Sie ein `tf`-LTI-Modell G1 mit der Übertragungsfunktion $G1 = \frac{2}{s^2+s+2}$

b. Erstellen Sie ein `zpk`-LTI-Modell G2 mit der Übertragungsfunktion $G2 = 6\frac{1-s}{(s+2)(s+3)}$

c. Erstellen Sie ein `ss`-LTI-Modell SS1 mit den Matrizen $A = [1, 2; -3, -4]$, $B = [1; 1]$, $C = [.2, 0]$, $D = 0$

d. Vergleichen Sie die Sprungantworten aller Systeme in einer einzigen Abbildung.

6.2 ☞ *Zeitdiskrete LTI-Modelle:*

a. Erstellen Sie ein `tf`-LTI-Modell dG1 mit der Übertragungsfunktion $dG1 = \frac{2}{z^2+0{,}3z-0{,}1}$ und Abtastzeit $1s$.

b. Wandeln Sie G2 aus Aufgabe 6.1 in ein zeitdiskretes Modell um. Dem Modellnamen soll hierbei ein d vorangestellt werden. Die Abtastzeit beträgt $0,1s$.

c. Vergleichen Sie die Sprungantworten von G2 und dG2 in einer figure. Ist die Abtastzeit ausreichend klein?

d. Wandeln Sie G2 in ein zeitdiskretes Modell fdG2 mit einer Abtastzeit von $1s$ um und vergleichen Sie die Sprungantworten der drei Systeme. Ist die Abtastfrequenz für fdG2 richtig? Warum?

e. Plotten Sie die Bode-Diagramme der drei Systeme und begründen Sie die Auswahl der besten Diskretisierung (dG2 oder fdG2?)

6.3 ☞ *Verkopplung von LTI-Modellen:*

a. Fassen Sie die LTI-Modelle G1 aus Aufgabe 6.1 und $G3 = \frac{s+1}{s+2}$ zu einem LTI-Modell GS (S wie Strecke) zusammen. Ist die Strecke stabil? Hat das System einen statischen Verstärkungsfaktor von 1?

b. Erstellen Sie einen PI-Regler GPI mit den Parametern $Ki = 3$ und $Kp = 3,75$.

c. Berechnen Sie die Übertragungsfunktion des geschlossenen Regelkreises GC und überprüfen Sie anhand eines PN-Bildes, ob er stabil ist.

d. Plotten Sie die Sprungantworten von GS und GC zusammen. Ist das System durch die Regelung schneller geworden? Ist die stationäre Regelabweichung noch vorhanden? Warum?

6.4 ☞ *Modelleigenschaften:* Ermitteln Sie für die einzelnen LTI-Modelle aus Aufgabe 6.1 die folgenden Werte:

a. Statische Verstärkung

b. Eigenwerte

c. Nullstellen

d. Natürliche Frequenzen und Dämpfungen

e. Beruhigungszeit und Endwert der Ausgangsgröße

6.5 ☞ *Systemantwort im Zeitbereich:* Zeigen Sie für die LTI-Modelle aus Aufgaben 6.1 und 6.2 folgendes an:

a. Anfangswertantwort von SS1 für $x(0) = [1; 1]$

b. Impulsantworten von G1 und G2 und von G2 und dG2

c. Zeigen Sie die Wurzelortskurven von G1 und G2 (zusammen)

d. Null-Polstellen-Verteilung von G1 und G2 (zusammen)

e. Null-Polstellen-Verteilung von G2 und dG2 (zusammen). Warum taucht hier ein Kreis auf?

f. Antwort von G2 auf den Eingang $\begin{cases} u = 0 & \forall \;\; t < 0s \\ u = \frac{t^2}{400} & \forall \;\; 0s \leq t \leq 20s \\ u = 1 & \forall \;\; 20s < t < 30s \\ u = 0 & \forall \;\; t \geq 30s \end{cases}$

6.6 ☞ *Systemantwort im Frequenzbereich:* Zeigen Sie für die LTI-Modelle aus Aufgaben 6.1 und 6.2 folgendes an:

a. Bode-Diagramm von G2, Amplitudenreserve und Phasenreserve (in einer Grafik).

b. Nyquist-Diagramm von G2, Ampitudenreserve und Phasenreserve (in einer Grafik).

c. Erzeugen Sie $G4 = G2 \cdot e^{-s}$ (Totzeit $1s$) und stellen Sie das zugehörige Nyquist-Diagramm dar. Was sind die Amplituden- und Phasenresreve?

d. Natürliche Frequenzen und Dämpfung

e. Beruhigungszeit und Endwert der Ausgangsgröße

6.7 ☞ *LTI-Viewer:* Analysieren die LTI-Modelle aus Aufgaben 6.1 und 6.2 mit Hilfe des Befehls `ltiview(G2,dG2)` und bearbeiten Sie folgende Punkte:

a. Lassen Sie sich den Höchstwert der Sprungantwort berechnen

b. Schauen Sie sich die Impulsantwort der Systeme an

c. Erzeugen Sie das Nyquist-Diagramm und zeigen Sie hier die Amplitudenreserve des Systems

d. Schalten Sie den Hintergrundraster ein

e. Lassen Sie sich jetzt die Sprungantwort und das Bode-Diagramm in zwei getrennten Grafiken darstellen.

6.8 ☞ *SISO-Tool:* Eine Strecke ist durch die Übertragungsfunktion $G_p = 9\frac{s+1}{(s-2)\,(s-5)}$ beschrieben. Bearbeiten Sie folgende Punkte:

a. Starten Sie das SISO-Tool und laden Sie dabei die Strecke

b. Lassen Sie sich die Wurzelortskurve und das Bode-Diagramm darstellen.

c. Schalten Sie das Hintergrundraster im Bode-Diagramm ein. Was sind die Amplituden- und Phasenreserven? (Hier soll der P-Regler eine Verstärkung von 1 besitzen.)

d. Stellen Sie die Sprungantworten des offenen und des geschlossenen Regelkreises in zwei Grafiken dar. Ist die Strecke stabil? Ist der geschlossene Regelkreis mit $Kp = 1$ stabil?

e. Bewegen Sie mit dem Hand-Tool die Pole des geschlossenen Regelkreises in der Wurzelortskurve so, dass das System stabil ist, nicht schwingt und beide Pole am gleichen Ort liegen. Was ist der dafür notwendige Wert von Kp? Betrachten Sie die Sprungantwort.

f. Lassen Sie sich die Anforderung an das System darstellen, sodass der Dämpfungsgrad $D = 0,707$ ist. Platzieren Sie die Pole, damit die Anforderung genau erfüllt wird. Was ist der Wert von Kp? Betrachten Sie die Sprungantwort.

g. Das System mit dem P-Regler ist nicht stationär genau (sehen Sie die Sprungantwort $y(t \to \infty) \neq 1$). Fügen Sie eine Polstelle so in den Kreis ein, dass das System stationär genau werden kann. Kann das System mit diesem zusätzlichen Pol und NUR einem Parameter K stabilisiert werden? Warum?

h. Fügen Sie eine weitere Nullstelle so in das System ein, dass Sie eine PI-Struktur $Gc = Kp + \frac{Ki}{s}$ erhalten. Probieren Sie verschiedene Positionen für die Nullstelle aus (auch in der rechten s-Halbebene). Wo wäre es sinnvoll, die Nullstelle zu platzieren? Warum?

i. Lassen Sie die Nullstelle in $s = -4$ und stellen Sie $Ki = 40$ ein. Betrachten Sie die Sprungantwort. Ist das System stationär genau? Warum?

6.9 ☞ *Entwurf in der Zustandsraumdarstellung:* Mit dem LTI-Zustandsraummodell SS1 aus Aufgabe 6.1 bearbeiten Sie folgende Punkte:

a. Lassen Sie sich die Rückführungsmatrix **K** berechnen, sodass die neue Lage der Polstellen $s = -4 \pm 4i$ ist. Erzeugen Sie das entsprechende LTI-Modell SS2.

b. Plotten Sie die Sprungantworten von SS1 und SS2. Ist SS2 schwingungsfähig? Warum? Sind die Systeme stationär genau? Überlegen Sie sich, wie Sie stationäre Genauigkeit erreichen können.

c. Skalieren Sie den Eingang von SS1 und SS2, sodass beide Systeme stationär genau werden. Speichern Sie diese Modelle als SS3 und SS4.

d. Plotten Sie die Sprungantwort von SS3 und SS4. Beurteilen Sie die Regelgüte von SS4 im Vergleich zu der des skalierten Originalsystems SS3.

e. Entwerfen Sie einen LQR-Regler für SS1 mit den Gewichtungsmatrizen `Q=diag([1 1])`, `R=1` und `N=0` und skalieren Sie den Eingang, um das System stationär genau zu machen. Speichern Sie das System als SS5.

f. Entwerfen Sie einen LQR-Regler für SS1 mit den Gewichtungsmatrizen `Q=diag([10 10])`, `R=1` und `N=0` und skalieren Sie den Eingang, um das System stationär genau zu machen. Speichern Sie das System als SS6.

g. Entwerfen Sie einen LQR-Regler für SS1 mit den Gewichtungsmatrizen `Q=diag([1 1])`, `R=10` und `N=0` und skalieren Sie den Eingang, um das System stationär genau zu machen. Speichern Sie das System als SS7.

h. Plotten Sie die Sprungantworten von SS5, SS6 und SS7. Analysieren Sie das Ergebnis und überlegen Sie sich den Effekt der Gewichtungsmatrizen **Q** und **R**.

6.10 ☞ *Die schwebende Kugel:* Eine Stahlkugel soll in folgender Anordnung durch einen Elektromagneten im Schwebezustand gehalten werden.

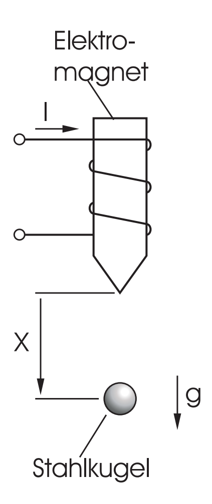


Abbildung 6.8: Skizze der Anordnung

Die vom Elektromagnet auf die Kugel ausgeübte Kraft F_m ist in erster Näherung durch folgende Gleichung gegeben:

$$F_m = a_1 \cdot \frac{I^2}{(a_0 + x)^2}$$

mit $a_0 = 6.9\ mm$ und $a_1 = 1.16 \cdot 10^{-4}\ N\, m^2/A^2$.

Die Masse der Kugel beträgt $m = 0.114\ kg$ und die Erdbeschleunigung $g = 9.81\ m/s^2$.

a. Stellen Sie die Differentialgleichung auf, die die Bewegung der Kugel im Magnetfeld beschreibt.

b. Welcher Strom I_A muss eingestellt werden, um die Kugel in einem Arbeitspunkt $X_A = 0.02$m zu halten?

c. Führen Sie für diesen Arbeitspunkt (X_A, I_A) eine Linearisierung der DGL durch und bringen Sie das System in Zustandsraumdarstellung. Geben Sie zusätzlich den Frequenzgang des Systemes an.

d. Ist das System stabil? Begründen Sie Ihre Antwort anhand der Physik, der DGL und der linearen Systemdarstellungen. Bestätigen Sie Ihre Antwort, indem Sie das nichtlineare System mit Simulink simulieren.

e. Finden Sie einen linearen Regler (P, PD, PID, PI), so dass das System stabilisiert wird. Überprüfen Sie Ihre Wahl anhand der Simulation.

6.11 ☞ *Der Van-der-Pol-Oszillator:* Der Van-der-Pol-Oszillator ist ein schwingungsfähiges System mit nichtlinearer Dämpfung und Selbsterregung. Für kleine Amplituden ist die Dämpfung negativ (die Amplitude wird vergrößert). Ab einem bestimmten Schwellenwert der Amplitude wird die Dämpfung positiv, das System stabilisiert sich und geht in einen Grenzzyklus über.

- Erstellen Sie ein Modell (vgl. Abb. 6.9) der nichtlinearen van-der-Pol[1]-Differentialgleichung

$$\frac{d^2x}{dt^2} + \mu \cdot \left(x^2 - 1\right) \cdot \frac{dx}{dt} + \omega^2 \cdot x = 0$$

  wobei $\mu = 1$, $\omega = 1$, $x(0) = 1$ und $\dot{x} = 0$.

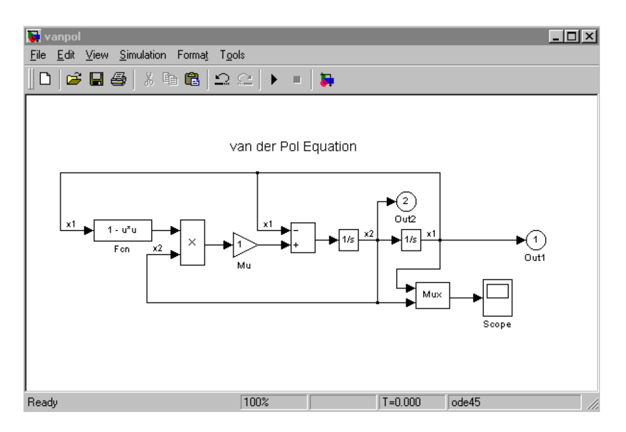


Abbildung 6.9: Modell der van-der-Pol-Differentialgleichung

  Speichern Sie es unter dem Namen **vanderpol**. Verwenden Sie die Blöcke Sum, Product, Fcn, Gain, Integrator, Mux, Scope und Outport. Achten Sie beim Erstellen auch darauf, bei welchen Blöcken der Name und wenn ja, welcher, angezeigt wird. Dieses Modell ist wie viele andere bei den *Examples* und *Demos* von MATLAB/SIMULINK hinterlegt.

- Benutzen Sie die Hilfe-Funktion um `x1` im Integrierer zwischen -1 und 1 zu begrenzen. Simulieren Sie das Ergebnis.

- Schalten Sie die Begrenzung wieder aus. Benutzen Sie erneut die Hilfe-Funktion, um `x1` im Integrierer zurückzusetzen (reset), sobald $x_2 = \dot{x}$ einen Nulldurchgang mit positiver Steigung besitzt. Simulieren Sie das Ergebnis.

- Ändern Sie die Formatierungen wie Farben, Schriftgrößen, ...

6.12 ☞ *Linearisierung in Simulink:* Gegeben ist das System $\dot{x} = -\sin(x) + u$. Hierbei ist u ein Sprung von 2 auf 0 bei $1s$. Bearbeiten Sie folgende Punkte:

a. Erstellen Sie das Modell in Simulink

b. Finden Sie das Gleichgewicht, wenn $u = 0$ ist. Nehmen Sie diesen Punkt als Arbeitspunkt und linearisieren Sie das System per Hand.

c. Lassen Sie sich von Simulink die Linearisierung berechnen und exportieren Sie das erhaltende Modell als GL.

[1] Balthazar van der Pol (1889-1959): Holländischer Elektroingenieur

d. Vergleichen Sie Ihre Linearisierung mit der von Simulink gelieferten. Sind diese gleich?

e. Fügen Sie dem Modell einen LTI-Block hinzu und laden Sie das linearisertes Modell. Vergleichen Sie die Ausgangsgrößen beider Systeme in einem Scope. Sind die Verläufe ähnlich?

f. Erhöhen Sie den Sprung: von 4 auf 0 statt von 2 auf 0. Simulieren Sie erneut und vergleichen Sie die Ausgänge. Was ist passiert und warum?

6.13 ☞ *Regelung eines Segway:* In dieser Übung soll mit Hilfe von Matlab ein (LQR-)Regler entworfen werden, der einen Segway im Gleichgewicht hält. Dafür sind folgende Schritte notwendig:

- Ermittlung der Regel- und der Stellgröße aus den Bewegungsgleichungen
- Linearisierung der Gleichungen
- Bestimmung der Zustandsrückführungsmatrix

Ermittlung der Regel- und der Stellgröße aus den Bewegungsgleichungen

Gegeben sind die Bewegungsgleichungen, denen ein Modell bestehend aus einer Achse mit zwei Rädern und einem Balken zugrunde liegt.

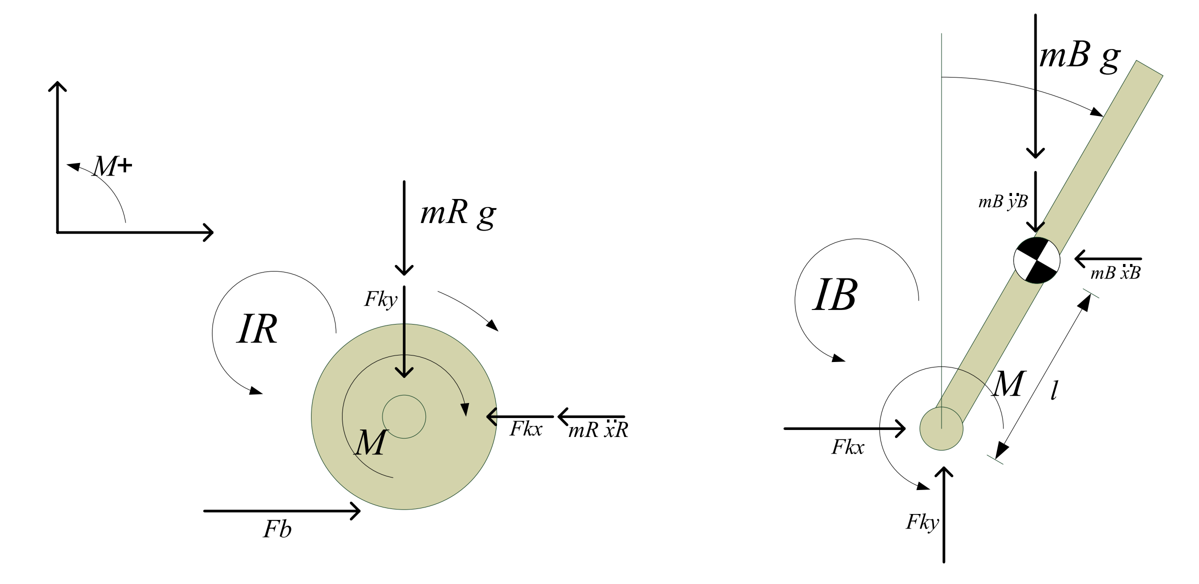


Abbildung 6.10: Skizze

Kräfte- und Momentengleichgewicht für das Rad ergeben folgende Gleichungen:

$$\Sigma F_X = 0: \qquad F_B - F_{KX} - m_R \cdot \ddot{x}_R = 0 \tag{1}$$

$$\Sigma M = 0: \qquad -M + F_B \cdot r + I_R \cdot \ddot{\alpha} = 0 \tag{2}$$

Für den Balken ergibt sich:

$$\Sigma F_X = 0: \qquad F_{KX} - m_B \cdot \ddot{x}_B = 0 \tag{3}$$

$$\Sigma F_Y = 0: \qquad F_{KY} - m_B \cdot \ddot{y}_B - m_B \cdot g = 0 \tag{4}$$

$$\Sigma M = 0: \qquad F_{KX} \cdot l \cdot cos\varphi - F_{KY} \cdot l \cdot sin\varphi + I_B \cdot \ddot{\varphi} + M = 0 \tag{5}$$

Kinematik des Rades:

$$x_R = r \cdot \alpha \tag{6}$$

Kinematik des Balkens:

$$x_B = r \cdot \alpha + l \cdot sin\varphi \tag{7}$$

$$y_B = l \cdot cos\varphi \tag{8}$$

a. Bestimmen Sie hieraus die Formeln für $\ddot{\varphi}$ und $\ddot{\alpha}$ in Abhängigkeit von $\alpha, \dot{\alpha}, \varphi, \dot{\varphi}, M$ und den Konstanten. Eliminieren Sie alle Kräfte und alle translatorischen Bewegungsgrößen.

Linearisierung der Gleichungen

Viele Regelungstechnische Methoden basieren auf einer Zustandsraumdarstellung. Dabei wird ein System durch geeignete 'Zustände' dargestellt. Für unseren 2D-Fall sind das z.B. die Winkel von Rad und Balken sowie die Winkelgeschwindigkeiten. Das Verhalten selbst wird dann dadurch beschrieben, dass Gleichungen für die Änderung dieser Zustände in Abhäningkeit von sich selbst und zusätzlichen Eingangsgrößen (u) zur Verfügung gestellt werden.

$$\begin{aligned} \dot{x}_1 &= f_1(x_1(t); \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \ldots &\vdots \ldots \\ \dot{x}_n &= f_n(x_1(t); \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \end{aligned}$$

oder als Vektordifferentialgleichung

$$\dot{\mathbf{X}} = \mathbf{f}(\mathbf{X}(t), \mathbf{U}(t))$$

Sind nicht alle Zustände des Systems Ausgänge oder werden die Ausgänge unmittelbar von den Eingängen beeinflusst, so berechnet man die Ausgangsgröße durch die Ausgangsgleichung

$$\mathbf{Y} = \mathbf{g}(\mathbf{X}(t), \mathbf{U}(t))$$

Aus dieser Darstellung kann eine lineare (vereinfachte) Form erzeugt werden, indem die Taylorreihe für diese Funktionen nach dem ersten Term abgebrochen wird.

$$\dot{X} = f(X,U) = f(X_0,U_0) + \underbrace{\left.\frac{\partial f}{\partial X}\right|_{X=X_0,U=U_0}}_{A} \cdot \delta X + ... + \underbrace{\left.\frac{\partial f}{\partial U}\right|_{X=X_0,U=U_0}}_{B} \cdot \delta U + ...$$

$$Y = g(X,U) = g(X_0,U_0) + \underbrace{\left.\frac{\partial g}{\partial X}\right|_{X=X_0,U=U_0}}_{C} \cdot \delta X + ... + \underbrace{\left.\frac{\partial g}{\partial U}\right|_{X=X_0,U=U_0}}_{D} \cdot \delta U + ...$$

Rechnet man für den stationären Fall und mit Abweichungsgrößen kommt man auf die Form

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$
$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$$
$$\text{mit } \mathbf{x} = X - f(X_0)\ ,\ \mathbf{y} = Y - Y_0 \text{ und } \mathbf{u} = U - U_0$$

In unserem Fall mit 4 Zustandgrößen ergeben sich die Matrizen aus

$$A = \left.\frac{\partial f}{\partial x}\right|_{x=x_0,u=u_0} = \texttt{jacobian}(\mathbf{f},\mathbf{x}) \quad B = \left.\frac{\partial f}{\partial u}\right|_{x=x_0,u=u_0} = \texttt{jacobian}(\mathbf{f},\mathbf{u})$$
$$C = \left.\frac{\partial f}{\partial x}\right|_{x=x_0,u=u_0} = \texttt{jacobian}(\mathbf{g},\mathbf{x}) \quad D = \left.\frac{\partial g}{\partial u}\right|_{x=x_0,u=u_0} = \texttt{jacobian}(\mathbf{g},\mathbf{u})$$

In Bild 6.11 ist der Signalfluß schematisch gezeigt.

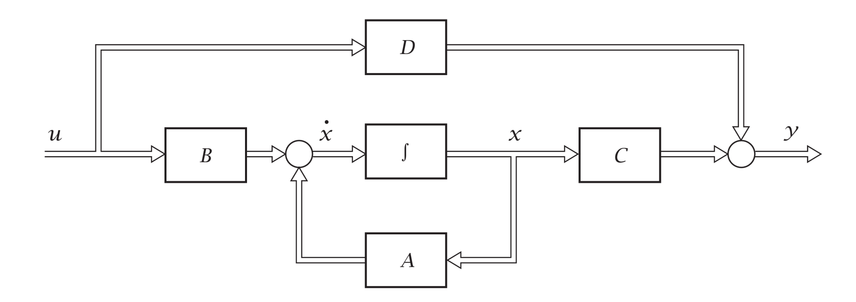


Abbildung 6.11: Signalfluss Lineare Zustandsraumdarstellung

b. Bringen Sie die zuvor berechneten Bewegungsgleichungen nun in folgende Zustandsraumform.

$$\begin{pmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \end{pmatrix} = \begin{pmatrix} f_1(x1,x2,x3,x4,u) \\ f_2(x1,x2,x3,x4,u) \\ f_3(x1,x2,x3,x4,u) \\ f_4(x1,x2,x3,x4,u) \end{pmatrix} \text{ mit } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \alpha \\ \varphi \\ \dot{\alpha} \\ \dot{\varphi} \end{pmatrix} \text{ und } u = M$$

c. Berechnen Sie nun die Matrizen A und B durch Linearisieren der Zustandsraumgleichungen. (C und D brauchen nicht berechnet zu werden. Da wir $\mathbf{x}$ als Ausgangsgröße verwenden (also $\mathbf{y} = \mathbf{x}$), wird C zur Einheitsmatrix und D zu 0.)

Bestimmung der Zustandsrückführungsmatrix

Eine Möglichhkeit der Reglerauslegung ist der sog. LQR-Regler (least quadratic Regulator). Hierbei lässt sich eine Zustandsrückführmatrix K bestimmen (Abb. 6.12), die zur Regelung des Systems aus einem beliebigen Zustand zu Null führt. Durch die Matrixform von K wird so eine Eingangsgröße (hier z.B. Drehmoment) aus einer Linearkombination der Zustände gebildet.

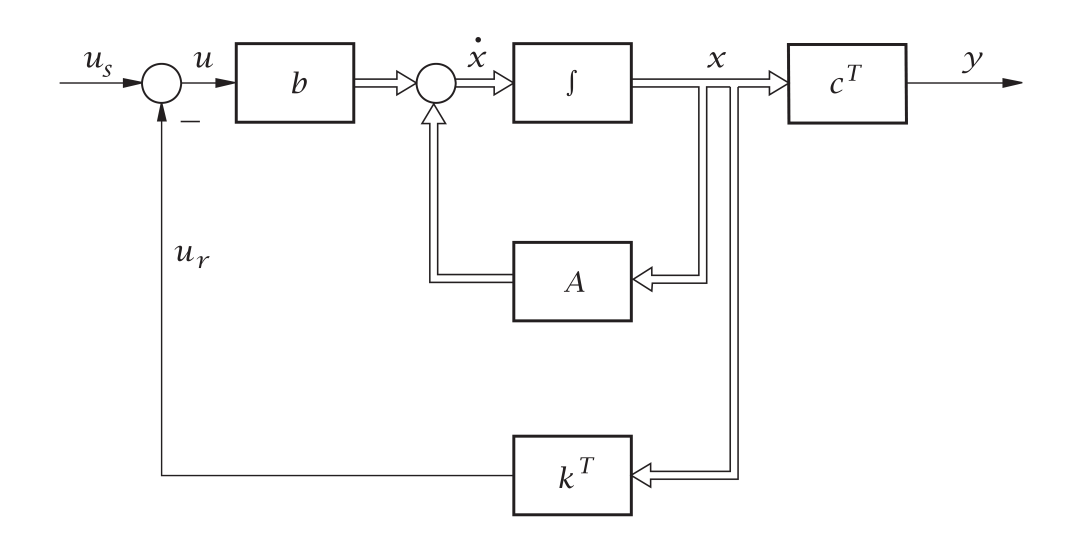


Abbildung 6.12: Zustandrückführung

Hierbei wird K so bestimmt, dass das 'Strafmaß'

$$J(u) = \int_0^\infty (x^T Q x + u^T R u) dt$$

das über den Regelungsverlauf integriert wird, minimal wird.

Q und R sind sog. 'Wichtungsmatrizen'. So bestimmt die Matrix Q, wie stark eine Abweichung der Zustandsgrößen bewertet wird und R, wie stark die Stellgrößen bestraft werden. Das bedeutet zum Beispiel, dass bei wachsenden Werten in der Matrix R die M resultierenden Momente kleiner werden.

In der Regel werden für Q und R Diagonalmatrizen verwendet, da diese auch interpretierbar sind. Beispiel:

$$\widetilde{Q} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\varphi} \\ \dot{l} \end{matrix} \tag{6.3}$$

$$R = \begin{pmatrix} 50 & 0 \\ 0 & 50 \end{pmatrix} \begin{matrix} M1 \\ M2 \end{matrix}$$

Die Berechnung von K lässt sich nur numerisch bewerkstelligen. In Matlab lässt sich die Matrix K über den Befehl `lqr` berechnen.

d. Berechnen Sie die Matrix K und simulieren Sie den geschlossenen Regelkreis.

7 GUI Programmierung

MATLAB bietet dem Anwender die Möglichkeit eigene grafische Benutzeroberflächen zu erzeugen. Mit Hilfe eines solchen **GUI** (**G**raphical **U**ser **I**nterface) gestaltet sich die Anwendung eines Programms sehr viel komfortabler als durch die manuelle Eingabe von Befehlen im Command Window.

7.1 GUIDE

Um GUIs zu erzeugen, bietet MATLAB einen Editor, den sog. **GUIDE**. Durch die Eingabe von `guide` im Command Window wird dieser gestartet. Im darauf folgenden Dialog lässt sich nun ein bestehendes GUI öffnen oder ein neues GUI aus einer Vorlage (template) erzeugen. Standardmäßig ist die Erzeugung eines leeren GUIs (Blank GUI) vorselektiert.
GUIDE erzeugt beim Speichern eine .fig-Datei und eine .m-Datei. In der .fig-Datei wird das Layout des erzeugten GUI gespeichert, in der .m-Datei die Funktionalität (siehe Abschnitt 7.2).

Formulareditor

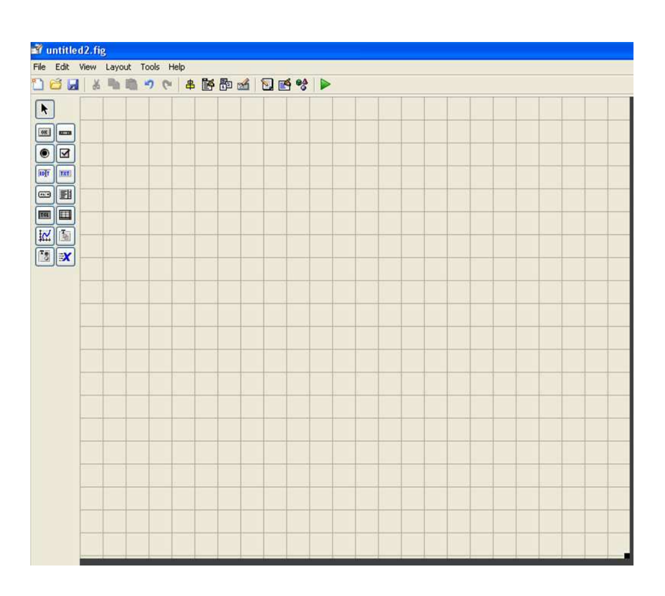


Abbildung 7.1: Das neue GUI

Abbildung 7.1 zeigt den Formulareditor eines leeren GUIs. Auf der linken Seite befinden sich die einzelnen Elemente, aus denen ein GUI zusammengesetzt werden kann:

- **Push Button**
  Ein Push Button ist eine Schaltfläche mit einer Beschriftung. Bei Betätigung der Schaltfläche wird eine hinterlegte Funktion ausgeführt.

- **Slider**
  Ein Slider ist ein Schieberegler mit dem Werte in einem Intervall mit einer bestimmten Schrittweite verändert werden können.

- **Radio Button**
  Ein Radio Button wird zur Auswahl von Optionen verwendet, die sich gegenseitig ausschließen. Wenn Radio Buttons in einer **Button Group** angeordnet werden, kann jeweils nur einer ausgewählt sein.

- **Check Box**
  Check Boxen werden zur Auswahl von Optionen verwendet, die in beliebiger Kombination ausgewählt werden können.

- **Edit Text**
  Edit Text ist ein Feld in dem Text oder Zahlen eingegeben werden können.

- **Static Text**
  Static Text dient dazu Text anzuzeigen, der nicht durch den Benutzer editiert werden kann, beispielsweise für Beschriftungen.

- **Pop Up Menu**
  Ein Pop Up Menu dient ähnlich wie Radio Buttons dazu, aus einer Auswahl von Optionen auszuwählen, die sich gegenseitig ausschließen. Da die Optionen eines Pop Up Menus nur während des Auswahlvorgangs eingeblendet werden, lassen sich damit auch umfangreiche Auswahllisten realisieren.

- **Listbox**
  Eine Listbox zeigt eine Liste von Elementen an, die in beliebiger Kombination ausgewählt werden können. Listenelemente können zur Laufzeit des Programms angehängt und entfernt werden.

- **Toggle Button**
  Ein Toggle Button ist ein Schalter, der durch einen Klick aktiviert und durch einen weiteren Klick wieder deaktiviert werden kann.

- **Table**
  Mit einer Table können Daten in tabellarischer Form angezeigt und editiert werden. Das bietet sich besonders bei Matrizen an.

- **Axes**
  Mit Axes lassen sich Plots oder Grafiken im GUI anzeigen.

- **Panel**
  In einem Panel können einzelne GUI-Elemente unter einer gemeinsamen Überschrift zusammengefasst werden. Dadurch wird die Übersichtlichkeit von GUIs mit vielen Bedienelementen erhöht.

- **Button Group**
  Zusammengehörige Radio Buttons werden in einer Button Group zusammengefasst. Es ist immer nur ein Radio Button in einer Button Group aktiviert.

Bedienelemente werden auf der GUI plaziert, indem zunächst das gewünschte Element in der linken Leiste ausgewählt wird und dann mit der linken Maustaste in das GUI eingefügt wird. Durch Ziehen mit gedrückter linker Maustaste lässt sich die Größe der Bedienelemente variieren.
Um einzelne Elemente bündig nebeneinander oder untereinander anzuorden, kann der Dialog

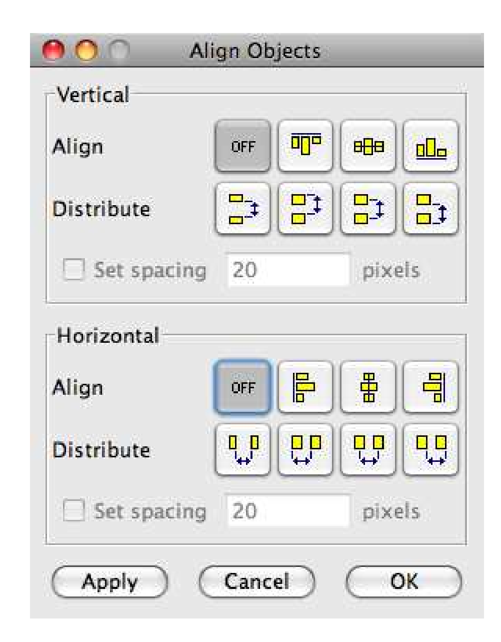


Abbildung 7.2: Elemente automatisch ausrichten

Tools → *Align Objects...* verwendet werden (Abb. 7.2). Mit diesem Dialog können ebenfalls feste Abstände zwischen den Bedienelementen angegeben werden (`Set Spacing`). Um Elemente automatisch anzuordnen, müssen diese zunächst markiert werden. Mehrere Elemente können entweder durch Gedrückthalten der Shift-Taste, oder durch aufspannen eines Selektierungsrahmens mit der linken Maustaste markiert werden.

Neben diesen Bedienelementen können dem GUI auch noch eine **Menu Bar**, **Context Menus** und eine **Toolbar** hinzugefügt werden. Eine Menu Bar befindet sich am oberen Fensterrand und enthält meistens Menüpunkte wie *File, Edit* und *View*. Eine Toolbar ist eine Leiste mit kleinen Symbolen direkt unter der Menu Bar. Context Menus werden mit einem Rechtlick angezeigt und beziehen sich immer auf das Element auf dem der Rechtsklick ausgeführt wurde.
Menu Bar und Context Menus werden mit Hilfe des *Menu Editor* (*Tools* → *Menu Editor...*) erzeugt und bearbeitet, die Toolbar mit Hilfe des *Toolbar Editors* (*Tools* → *Toolbar Editor...*).

Property Inspector

Die Eigenschaften der Bedienelemente können mit Hilfe des **Property Inspectors** bearbeitet werden (siehe Abb. 7.3). Gestartet wird der Property Inspector durch den Menüpunkt *Property Inspector* im Kontextmenü des jeweiligen Bedienelements oder über *view* → *Property Inspector*.
Zu den Eigenschaften, die mit Hilfe des Property Inspectors bearbeitet werden können, zählen Größe, Farbe und Position des Elements. Bei Bedienelementen, die mit Text versehen sind (z.B. Push Button, Edit Text etc.), kann der angezeigte Text, die verwendete Schriftart und die Schriftgröße verändert werden.
Mit der Eigenschaft *Tag* kann der Name des Bedienelements verändert werden unter dem es aus MATLAB erreichbar ist. Dabei ist zu beachten, dass die Callback-Funktionen von Hand an den neuen *tag* angepasst werden müssen (siehe Abschnitt 7.2).
Die Eigenschaften werden beim Schließen des Property Inspectors automatisch übernommen und gespeichert.

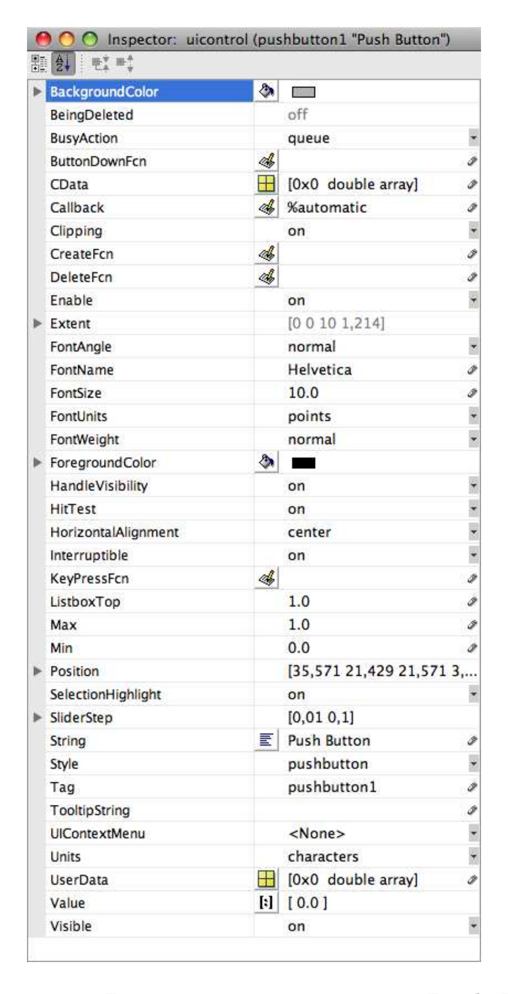


Abbildung 7.3: Property Inspector eines Push Buttons

Aufgaben und Übungen

7.1 ☞ *Erstellen eines GUI*: Erstellen Sie ein GUI, dass aus drei Textfeldern und einem Push Button besteht. Erzeugen sie eine große Überschrift 'Addierer' und beschriften Sie die Textfelder mit '1. Summand', '2. Summand' und 'Summe'. Verwenden Sie dazu *Static Text*. Ändern Sie die Beschriftung des Push Buttons in 'Addiere!'.

7.2 Callbacks

Callbacks sind MATLAB-Funktionen, die aufgerufen werden, wenn der Benutzer mit dem GUI interagiert, z.B. mit der Maus auf einen Push Button klickt.

Standard-Callbacks

Callbacks können für jedes Bedienelement erzeugt werden, auch für Elemente mit denen der Benutzer nicht direkt interagieren kann, wie z.B. Static Text. Folgende Callbacks können für jedes Bedienelement erzeugt werden:

- *CreateFcn* wird ausgeführt, wenn das Bedienelement erzeugt wird.
- *DeleteFcn* wird ausgeführt, wenn das Bedienelement gelöscht wird.
- *ButtonDownFcn* wird ausgeführt, wenn die Maustaste über dem Bedienelement gedrückt wird.
- *KeyPressFcn* wird ausgeführt, wenn eine Taste auf der Tastatur gedrückt wird. Das entsprechende Bedienelement dazu markiert sein.

Bei Bedienelementen, mit denen der Benutzer interagieren kann, gibt es ein Standard-Callback, das einfach als *Callback* bezeichnet wird. Es wird ausgeführt, wenn der Benutzer mit dem Bedienelement interagiert (z.B. drücken eines Push Buttons oder Texteingabe eines Edit Text).
Das GUI-Fenster kann auch Callbacks auslösen, z.B. wenn der Benutzer die Größe des Fensters ändert oder das Fenster schließen will.

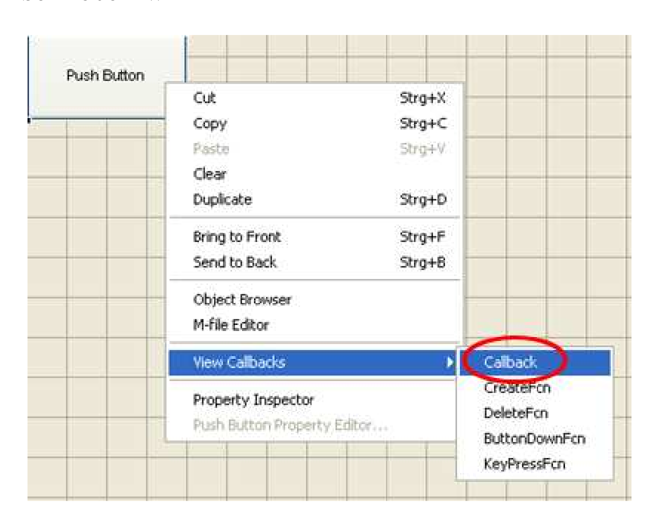


Abbildung 7.4: Callbacks eines Pushbuttons

Handles

Handles sind Zeiger über die auf ein Bedienelement zugegriffen werden kann. Jeder Callback-Funktion werden drei Parameter übergeben: *hObject, eventdata* und *handles*. *hObject* ist der Zeiger auf das Bedienelement selbst, welches den Callback ausgelöst hat und *handles* enthält alle Zeiger auf die übrigen Elemente des GUIs. *eventdata* dient nur als Platzhalter für eine Funktionalität, die in der aktuellen Version von MATLAB noch nicht implementiert ist.
Um die Eigenschaften eines Bedienelements zu setzen oder auszulesen, werden die Befehle `set` und `get` verwendet. Mit folgendem Befehl lässt sich der Inhalt eines Textfeldes in einer Variablen abzuspeichern:

```
>>text = get(handles.edit1,'String');
```

Mit diesem Befehl wird der Inhalt des Textfeldes mit dem Text `Neuer Text` überschrieben:

```
>>set(handles.edit1,'String','Neuer Text');
```

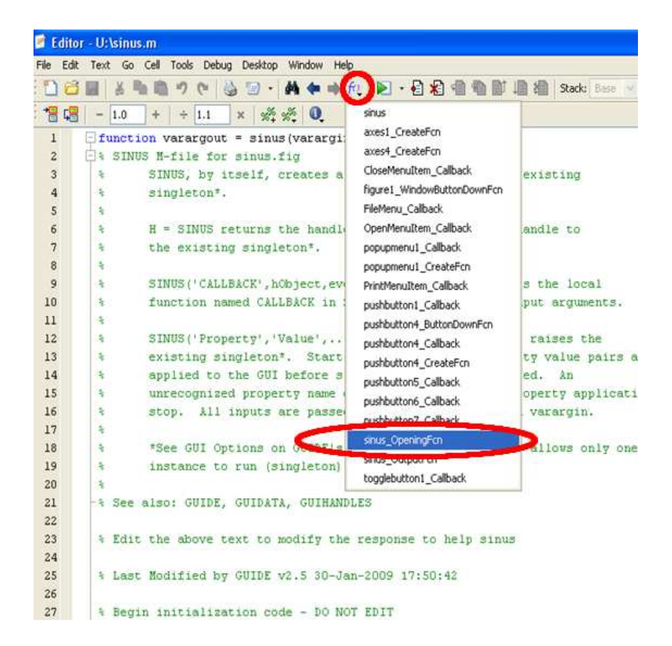


Abbildung 7.5: Auflistung aller Funktionen

Auf diese Weise lassen sich alle Eigenschaften aus dem Property Inspector beeinflussen.

Eine elegante Methode Daten zwischen Callback-Funktionen auszutauschen besteht darin die Variable `handles` zu verwenden. Bei `handles` handelt es sich nämlich um ein Struct. Structs können in MATLAB dynamisch mit eigenen Variablen erweitert werden. In der folgenden Zeile wird `handles` um die Variable `a` erweitert:

```
>>handles.a = 42;
```

Die veränderte Struktur muss nun noch gespeichert werden:

```
>>guidata(hObject, handles);
```

Da `handles` an alle Callback-Funktionen übergeben wird, ist nun auch die Variable `a` in allen Callback-Funktionen bekannt.

Aufgaben und Übungen

7.2 ☞ *Addition von Zahlen*: Verwenden Sie das GUI aus Aufgabe 7.1 und implementieren Sie die Funktionalität. *Hinweis:* Um diese Aufgabe zu lösen, müssen Sie nur ein Callback bearbeiten.

7.3 ☞ *Programmieren eines Pop-Up-Menüs*: Erstellen Sie ein GUI, dass nur ein Pop-Up-Menü und Achsen für ein Diagramm enthält. In dem Pop-Up-Menü soll es die Auswahlen **Membrane**, **Peaks** und **Close** geben. Bei einer der ersten beiden Auswahlen soll der jeweilige Befehl mit dem Namen ausgeführt werden und dann das Bild in dem vorgesehenen Diagramm anzeigen, bei **Close** soll das GUI geschlossen werden.

7.4 ☞ *Plot einer Funktion*: Erstellen Sie ein GUI, das eine **Radio Button Group** enthält, in der man eine Farbe auswählen kann. Es soll eine **axes** vorhanden sein, in der ein Plot durchgeführt

werden kann. In einem Eingabefeld soll noch eine Funktion eingegeben werden, die dann geplottet wird. Jedesmal, wenn eine Auswahl oder die Funktion geändert wird, soll der Plot auch aktualisiert werden.

7.5 ☞ *Altersberechnung*: Erstellen Sie ein GUI, das einen Static Text, einen Edit Text und einen Push Button enthält. Im Edit Text kann der Benutzer sein Geburtsdatum eingeben. Durch Drücken des Push Buttons soll das Alter im Static Text angezeigt werden. *Tipp:* `date`.

7.6 ☞ *Konvertierung eines Bildes*: Erstellen Sie ein GUI, welches ein Bild laden und anzeigen kann. Durch Drücken eines Push Buttons soll das Bild in ein Graustufen-Bild umgewandelt werden. *Tipp:* `imread, imshow, uigetfile, rgb2gray`.

7.3 GUIs ohne GUIDE

GUIs können in MATLAB auch ohne die Verwendung des grafischen Editors GUIDE erstellt werden. Das ist nützlich, um grafische Benutzeroberflächen dynamisch in Abhängikeit vorheriger Benutzereingaben oder Berechnungen zu generieren.
Zunächt wird mit dem Befehl `figure` ein neues Fenster geöffnet und anschließend mit der Funktion `uicontrol` die Bedienelemente platziert. Als Parameter weren der Funktion Wertepaare übergeben, die den Praperties aus dem Property Inspector entsprechen. Als Rückgabewert liefert die Funktion einen Zeiger auf das erzeugte Bedienelement.
Der folgende Aufruf erzeugt einen Push Button mit dem Text 'OK' an der Position x = 10 und y = 10 mit der Breite 50 und der Höhe 20. Die Positionsangaben sind jeweils in Pixel wobei sich der Punkt x = 0 und y = 0 an der linken unteren Ecke des Fensters befindet.

```
>> handle = uicontrol('style','pushbutton','String','OK', 'position',[10,10,50,20]);
```

Der Parameter `style` bestimmt welche Art Bedienelemt erzeugt wird. Möglich sind:

- `checkbox`
- `edit`
- `frame`
- `listbox`
- `popupmenu`
- `pushbutton`
- `radiobutton`
- `slider`
- `text`
- `togglebutton`

Mit dem Parameter `callback` kann beispielsweise einem Push Button ein, oder mehrere Befehle zugewiesen werden, die bei dessen Betätigung ausgeführt werden sollen. Dazu werden die Befehle einfach als String übergeben:

```
>>handle = uicontrol('style','pushbutton','callback','a=a+1; disp(a);', ...
```

Aufgaben und Übungen

7.7 ☞ *GUI ohne GUIDE*: Erzeugen Sie einen Addierer wie in Aufgabe 7.1 ohne GUIDE zu verwenden!

8 Objektorientierte Programmierung

Objektorientierung ist ein in der Programmierung weit verbreitetes Konzept, bei dem Daten und zugehörige Funktionen in Objekten zusammengefasst werden.

8.1 Grundbegriffe der objektorientierten Programmierung

In der objektorientierten Programmierung werden die verwendeten Daten klassifiziert. Das heißt, jedem Datentyp werden bestimmte Eigenschaften und zu ihm passende Funktionen zugeordnet. Wenn man zum Beispiel in einem Programm die Mitarbeiter eines Unternehmens verwalten möchte, kann man die *Klasse* `Mitarbeiter` definieren. Jeder Mitarbeiter ist dann ein *Objekt* dieser Klasse und hat die in dieser definierten *Eigenschaften* wie etwa seinen *Namen*, die *Abteilung*, in der er arbeitet, seinen direkten Vorgesetzten und seine *Gehaltsstufe*. Außerdem kann man in der Klasse Funktionen definieren, die nur im Zusammenhang mit Objekten der Klasse `Mitarbeiter` benötigt werden, wie zum Beispiel `befoerdere`, `versetzeNach` o.ä. Diese Funktionen einer Klasse heißen *Methoden*.

Ein wichtiges Konzept der objektorientierten Programmierung ist die *Vererbung*. Sie ermöglicht die einfache Erzeugung von Subklassen, die im Grunde eine Kopie der Originalklasse (der Superklasse) darstellen, die um eigene Eigenschaften und Funktionen erweitert werden können. So ist im obigen Beispiel etwa eine von der Klasse `Mitarbeiter` abgeleitete Klasse `Praktikant` denkbar. Diese hat dann die gleichen Eigenschaften wie ihre Superklasse, kann jedoch um nur für Praktikanten relevante Eigenschaften wie z.B. *LetzterArbeitstag* ergänzt werden.

Ein weiteres für die objektorientierte Programmierung typisches Konzept ist das der Datenkapselung. Darunter versteht man das Verbergen bestimmter Klassenelemente vor dem Zugriff von außerhalb der Klasse. Somit kann man zum Beispiel vorgeben, dass nur die Methoden eines Objekts auf dessen Eigenschaften zugreifen können.

8.2 Tutorial: die Klasse „Mitarbeiter“

Die Funktionen und die Eigenschaften, die ein Objekt haben soll, werden in einer Klasse definiert. Wie das funktioniert, soll anhand eines Beispiels demonstriert werden. Dafür wird die Klasse `Mitarbeiter` erzeugt, mit der die Mitarbeiter eines Unternehmens erfasst werden können. Jedes Objekt der Klasse soll über die Eigenschaften *Name*, *Gehaltsstufe* und *Gehalt* verfügen. Außerdem soll die Klasse Funktionen zur Verfügung stellen, mit denen die Datensätze erstellt, gelesen und verändert werden können.

Definition einer Klasse in Matlab

Zunächst erzeugt man das Grundgerüst einer Klassendefinition, indem man im MATLAB-Menü *File* → *New* → *Class M-File* auswählt. Somit wird im Editor eine Datei mit folgendem Inhalt geöffnet:

```
classdef Untitled
   %UNTITLED Summary of this class goes here
   %   Detailed explanation goes here
```

```
    properties
    end

    methods
    end

end
```

Nach dem Schlüsselwort `classdef` steht der Name der Klasse (hier: `Untitled`). Der Klassenname sollte möglichst aussagekräftig sein. Außerdem muss er mit einem Buchstaben beginnen und darf keine Umlaute enthalten.

In der folgenden Zeile steht eine Kurzbeschreibung in einem Kommentar. Diese erscheint unter *Details*, wenn die Klasse im Fenster *Current Folder* markiert wird.

Nach der Kurzbeschreibung steht eine ausführlichere Beschreibung, ebenfalls als Kommentar. Sowohl die Kurz- als auch die ausführliche Beschreibung erscheinen, wenn man im *Command Window* `help` *Klassenname* eingibt.

Unter `properties` werden die Eigenschaften der Klasse gesetzt. Hier können Variablennamen mit oder ohne Wertzuweisung stehen. Nach dem Schlüsselwort `methods` folgen die zur Klasse gehörenden Funktionen, die in diesem Zusammenhang *Methoden* genannt werden.

Benennen Sie die Klasse in `Mitarbeiter` um.

Eigenschaften der Klasse

Fügen Sie nun unter `properties` die Eigenschaften *Name* und *Gehaltsstufe* ein. Geben Sie der Gehaltsstufe einen Standard-Wert von 1. Speichern Sie die Datei nun ab. Hierbei ist zu beachten, dass der Dateiname dem Klassennamen (mit der Endung .m) entsprechen muss, in diesem Fall also `Mitarbeiter.m`.

```
classdef Mitarbeiter
  %MITARBEITER Klasse für die Verwaltung der Mitarbeiter
  % Hier könnte eine ausführlichere Erklärung stehen

    properties
        Name
        Gehaltsstufe = 1;
    end

    methods
    end

end
```

Nun können Sie mit

```
>> mitarbeiter1 = Mitarbeiter;
```

ein neues Objekt `mitarbeiter1` der Klasse `Mitarbeiter` erstellen. Auf die Eigenschaften des Objektes kann nun genauso wie auf die Elemente einer Structure zugegriffen werden, z.B.:

```
>> mitarbeiter1.Name = 'Max Mustermann';
>> mitarbeiter1.Gehaltsstufe = 2;
```

Der Vorteil eines Objekts gegenüber einer Structure wird deutlich, wenn dem Benutzer bei einer Wertzuweisung ein Tippfehler unterläuft. Im folgenden Beispiel sei `mitarbeiter2` eine Structure, dessen einzigem Feld *Gehaltsstufe* ein neuer Wert zugewiesen werden soll:

```
>> mitarbeiter2.Gehalsstufe = 4;
```

Wenn der Benutzer den Tippfehler in *Gehalsstufe* nicht sieht, wird er glauben, das Feld Gehaltsstufe hätte jetzt den Wert 4. Tatsächlich wurde aber ein neues Feld mit dem Wert 4 angelegt:

```
>> mitarbeiter2

mitarbeiter2 =

    Gehaltsstufe: 1
     Gehalsstufe: 4
```

Passiert dem Benutzer der gleiche Fehler bei einem Objekt, wird er sofort darauf hingewiesen:

```
>> mitarbeiter1.Gehalsstufe = 3;
??? No public field Gehalsstufe exists for class Mitarbeiter.
```

Methoden der Klasse

In diesem Abschnitt wird die Klasse um drei Methoden erweitert. Zunächst wird ein Konstruktor hinzugefügt.

Der Konstruktor

Ein Konstruktor ist eine Methode, die bei der Erzeugung eines neuen Objekts automatisch aufgerufen wird und dessen Eigenschaften initialisiert. Der Konstruktor hat immer den gleichen Namen wie die zugehörige Klasse und eine frei benennbare Objekt-Variable als Rückgabewert. Optional können dem Konstruktor Parameter übergeben werden.

```
classdef Mitarbeiter
    %MITARBEITER Klasse für die Verwaltung der Mitarbeiter
    %  Hier könnte eine ausführlichere Erklärung stehen

    properties
        Name
        Gehaltsstufe = 1;
    end

    methods
        function obj = Mitarbeiter(name,gs) % Konstruktor
```

```
            if nargin == 0
               obj.Name = 'unbekannt';
               disp('Bitte ändern Sie den Namen des Mitarbeiters.');
            else
               obj.Name = name;
               if nargin > 1
                  obj.Gehaltsstufe = gs;
               end
            end
         end

   end

end
```

Der Konstruktor der Klasse *Mitarbeiter* hat zwei Parameter. Bei seinem Aufruf überprüft er zunächst, ob wenigstens ein Name (erster Parameter) übergeben wurde[1]. Ist dies nicht der Fall, wird der Name auf 'unbekannt' gesetzt und der Benutzer gebeten, den Namen (manuell) zu ändern. Andernfalls wird der eingegebene Name der Eigenschaft *Name* zugewiesen und es wird überprüft, ob auch eine Gehaltsstufe (zweiter Parameter) übergeben wurde. Wenn ja, wird sie der entsprechenden Eigenschaft zugewiesen, andernfalls wird der im `properties`-Block eingestellte Standard-Wert von 1 verwendet. Natürlich könnte auch für den Mitarbeiter-Namen ein Standard-Wert im `properties`-Block eingestellt werden (statt im Konstruktor).

Weitere Methoden

Nun werden der Klasse zwei weitere Methoden hinzugefügt. Die erste (`zeigeAktuellesMonatsgehalt`) berechnet aus der Gehaltsstufe des Mitarbeiters sein Monatsgehalt. Mit der zweiten (`befoerdere`) wird die Gehalsstufe des Mitarbeiters um 1 erhöht.

```
classdef Mitarbeiter
   %MITARBEITER Klasse für die Verwaltung der Mitarbeiter
   %  Hier könnte eine ausführlichere Erklärung stehen

   properties
      Name
      Gehaltsstufe
   end

   methods
      function obj = Mitarbeiter(name,gs) % Konstruktor
         if nargin == 0 % nargin ist die Anzahl der Parameter
            obj.Name = 'unbekannt';
            disp('Bitte ändern Sie den Namen des Mitarbeiters.');
         else
            obj.Name = name;
```

[1] `nargin` gibt die Anzahl der vom Benutzer übergebenen Parameter zurück und kann in jeder Funktion verwendet werden.

```
                if nargin > 1
                    obj.Gehaltsstufe = gs;
                end
            end
        end
        function zeigeAktuellesMonatsgehalt(obj)
            gehalt = 1500 + obj.Gehaltsstufe * 300;
            disp(['Das aktuelle Monatsgehalt von ' obj.Name...
                ' beträgt ' num2str(gehalt) ' Euro.']);
        end
        function obj = befoerdere(obj)
            if obj.Gehaltsstufe < 5
                obj.Gehaltsstufe = obj.Gehaltsstufe + 1;
                disp('Beförderung ist erfolgt.');
            else
                disp('Beförderung nicht möglich!');
            end
        end
    end

end
```

Der Inhalt der Funktionen ist selbsterklärend. Zu beachten ist nur, dass die Funktion *befoerdere* die Objektvariable *obj* sowohl als Eingabe- als auch als Rückgabewert enthalten muss, da erst die alte Gehaltsstufe ausgelesen und dann die neue wieder hineingeschrieben wird.

Die Verwendung der Methoden kann auf zwei Arten erfolgen. Entweder wird das Objekt wie in der Funktionsdefinition als Parameter übergeben:

```
>> zeigeAktuellesMonatsgehalt(mitarbeiter1); % Syntax wie in üblichen m-Funktionen
```

Oder der Funktionsname wird ohne Parameter mit einem Punkt an den Objektnamen gehängt:

```
>> mitarbeiter1.zeigeAktuellesMonatsgehalt; % Syntax wie in C++
```

Da die Methode `befoerdere` das Objekt verändert, muss diesem der Rückgabewert der Methode auch zugewiesen werden. Also:

```
>> mitarbeiter1 = befoerdere(mitarbeiter1);
```

Oder:

```
>> mitarbeiter1 = mitarbeiter1.befoerdere;
```

Andernfalls erfolgt eine Meldung über eine erfolgreiche Beförderung im Command Window ohne dass die neue Gehaltsstufe im Objekt abgespeichert werden kann.

Der Destruktor

Der Destruktor ist wie der Konstruktor eine spezielle Methode. Er wird immer ausgeführt, wenn ein Objekt zerstört (d.h. gelöscht) wird. So kann zum Beispiel erreicht werden, dass eine geöffnete

Datei noch gespeichert wird, wenn MATLAB geschlossen wird und somit alle Objekte gelöscht werden. Anders als Konstruktoren sind Destruktoren kein zwingender Bestandteil einer Klasse. Konstruktoren können nur in Handle-Klassen erstellt werden. Diese werden im Abschnitt *Value- und Handle-Klassen* erläutert.

Eigenschafts-Attribute

Bisher enthält die Klasse `Mitarbeiter` nur den Standardfall der variablen, unabhängigen, von außen sichtbaren Eigenschaften. Eigenschaften können aber auch konstant, abhängig von anderen Eigenschaften und/oder von außen unsichtbar sein.

Konstante Eigenschaften

Soll eine Klasse konstante Eigenschaften enthalten, muss dafür ein eigener `properties`-Block erstellt werden. Hinter dem Schlüsselwort `properties` steht in diesem Fall das Wort `Constant` in Klammern. Nachfolgend wird die Klasse `Mitarbeiter` um eine konstante Eigenschaft erweitert:

```
    properties
        Name
        Gehaltsstufe = 1;
    end

    properties (Constant)
        MaximaleGehaltsstufe = 5;
    end
```

Wie das Wort `Constant` schon ausdrückt können die Werte konstanter Eigenschaften zwar gelesen aber nicht überschrieben werden.

Abhängige Eigenschaften

In einer Klasse können Eigenschaften definiert werden, deren Wert vom Wert einer (oder mehreren) anderen Eigenschaft(en) abhängt. Auch solche Eigenschaften werden in einem separaten `properties`-Block definiert. Dieser wird mit dem Zusatz `(Dependent)` versehen. Nun soll die Klasse `Mitarbeiter` um die von der Gehaltsstufe abhängige Eigenschaft *Gehalt* erweitert werden:

```
    properties (Dependent)
        Gehalt
    end
```

Dies ist nur sinnvoll, wenn auch definiert wird, in welcher Weise das Gehalt von der Gehaltsstufe abhängt. Dieser Zusammenhang wird in einer neuen Methode namens `get.Gehalt` festgelegt.

```
    methods
        function obj = Mitarbeiter(name,gs)
```

...

```
        end
        function gehalt = get.Gehalt(obj)
```

```
            gehalt = 1500 + obj.Gehaltsstufe * 300;
        end
```

...

```
    end
```

Die Funktion enthält die gleiche Formel zur Berechnung des Gehalts, die zuvor in der Funktion `zeigeAktuellesMonatsgehalt` verwendet wurde. Die Schreibweise *function [Rückgabevariable] = get.[Name der Eigenschaft]([Objektvariable])* ist zwingend notwendig. Der neue Wert wird immer erst berechnet, wenn der Benutzer oder eine interne Funktion die Eigenschaft *Gehalt* abfragt (dehalb auch das `get`).

Zugriffs-Attribute

Ein wichtiges Konzept der objektorientierten Programmierung ist die Datenkapselung. Darunter versteht man die Kontrolle über den Zugriff auf Eigenschaften und Methoden einer Klasse. In der Standard-Einstellung `public` kann man von außerhalb der Klasse auf entsprechende Eigenschaften und Methoden zugreifen. Wenn dies nicht erwünscht ist (z.B. weil man vor der Änderung einer Eigenschaft die Erfüllung bestimmter Bedingungen überprüfen möchte), kann man den Zugriff auf `private` oder `protected` setzen. `private` erlaubt nur den Zugriff durch Methoden derselben Klasse, `protected` erlaubt zusätzlichen Zugriff durch Methoden einer Subklasse. Für die gleichzeitige Beschränkung von Lese- und Schreibzugriff wird das Schlüsselwort `Access` verwendet. Möchte man nur den Schreib- oder nur den Lesezugriff (auf eine Eigenschaft) einschränken, verwendet man `SetAccess` bzw. `GetAccess` an Stelle von `Access`. Untenstehende Tabelle gibt einen Überblick über den standardmäßigen Schreib- und Lesezugriff mit den verschiedenen Zugriffs-Attributen.

	public	protected	private
Zugriff innerhalb der Klasse	ja	ja	ja
Zugriff aus einer Subklasse	ja	ja	nein
Zugriff von außerhalb der Klasse	ja	nein	nein

Nachfolgend wird der Schreibzugriff auf die *Gehaltsstufe* in der Klasse `Mitarbeiter` auf `private` gesetzt. Hierfür muss wieder ein eigener `properties`-Block verwendet werden:

```
    properties
        Name
    end

    properties (SetAccess = private)
        Gehaltsstufe = 1;
    end
```

Jetzt kann die Gehaltsstufe zwar noch vom Benutzer abgefragt, jedoch nicht mehr direkt verändert werden. Eine Veränderung ist nur noch über die Methode `befoerdere` möglich.

Sichtbarkeit von Eigenschaften

Wenn man sich einen schnellen Überblick über die Eigenschaften eines Objekts verschaffen möchte, kann man einfach den Namen des Objekts in das Command Window eingeben:

```
>> mitarbeiter1

mitarbeiter1 =

  Mitarbeiter

  Properties:
                    Name: 'Max'
            Gehaltsstufe: 3
    MaximaleGehaltsstufe: 5
                  Gehalt: 2400

  Methods
```

Jeder größer die Klasse wird, desto unübersichtlicher wird die Liste der Eigenschaften. Deshalb ist es in manchen Fällen wünschenswert, dass in solch einer Übersicht nur für den Benutzer relevante Eigenschaften auftauchen. Dies erreicht man, indem man entsprechende Eigeschaften mit dem Attribut `Hidden` versieht. In der Klasse Mitarbeiter braucht die maximale Gehaltsstufe nicht immer angezeigt werden. Deshalb erhält sie `Hidden` als zweites Attribut:

```
    properties (Constant, Hidden)
        MaximaleGehaltsstufe = 5;
    end
```

Mehrere Attribute werden also einfach durch Kommata getrennt aneinandergereiht. Auch im *Variable Editor* taucht die Eigenschaft *MaximaleGehaltsstufe* nicht mehr auf. Anders als bei der Verwendung von `Access = private` kann der Wert dieser Eigenschaft nach wie vor abgefragt werden:

```
>> mitarbeiter1.MaximaleGehaltsstufe

ans =

     5
```

Vererbung

Eine große Stärke der objektorientierten Programmierung ist das Konzept der Vererbung. So ist es möglich, eine Klasse zu schaffen, die zunächst einmal nur eine Kopie der ursprünglichen Klasse darstellt. Sie *erbt* die Eigenschaften und Methoden der Superklasse (auch Elternklasse genannt). Dann kann die neue Klasse beliebig um eigene Eigenschaften und Methoden erweitert werden (die der Superklasse dann nicht zur Verfügung stehen). Der Vorteil gegenüber einem tatsächlichen Kopieren des m-Codes der ursprünglichen Klasse in eine neue Klassendatei liegt in der besseren Wartbarkeit des objektorientierten Programms. Denn hier muss bei einer nachträglichen Änderung einer Methode oder einer Eigenschaft diese nur an einer Stelle angepasst werden. Für die abgeleitete(n) Klasse(n) gilt diese Änderung dann automatisch. Je größer ein Projekt wird, desto stärker macht sich dieser Vorteil bemerkbar.

Nachfolgend soll von der zuvor erstellten Klasse `Mitarbeiter` die Subklasse `Praktikant` abgeleitet werden. Erstellen Sie dazu im Verzeichnis der Klasse Mitarbeiter eine neue Klassendatei (MATLAB-Menü → *File* → *New* → *Class M-File*) und speichern Sie sie unter dem Namen *Praktikant.m* ab. Eine Vererbung erreicht man durch ein kleiner-als-Symbol (<) nach dem Klassennamen, gefolgt vom Namen der Superklasse. Erweitern Sie die Datei außerdem um die Eigenschaften *LetzterArbeitstag* und *WeitereZusammenarbeitErwuenscht*.

```
classdef Praktikant < Mitarbeiter
   %PRAKTIKANT Klasse für die Verwaltung der Praktikanten
   %   Detailed explanation goes here

   properties
      LetzterArbeitstag
      WeitereZusammenarbeitErwuenscht
   end

   methods
   end

end
```

Zusätzlich zu den neu definierten Eigenschaften hat die Klasse alle Eigenschaften und Methoden, die die Superklasse *Mitarbeiter* auch hat. Die einzige Ausnahme hiervon ist der Konstruktor. Damit auch dieser von der Superklasse übernommen wird, muss eine Konstruktor-Methode mit folgendem Inhalt eingefügt werden:

```
   methods
      function obj = Praktikant(name,gs)
         obj = obj@Mitarbeiter(name,gs);
      end
   end

end
```

Nun sollen aber Praktikanten immer die Gehaltsstufe 0 erhalten. Deshalb wird der Konstruktor entsprechend angepasst:

```
   methods
      function obj = Praktikant(name)
         obj = obj@Mitarbeiter(name,0);
      end
   end

end
```

Somit muss bei der Erstellung eines neuen Objekts nur noch der Name des Praktikanten als Parameter übergeben werden.

Überladen von Funktionen und Operatoren

Abgeleitete Klassen erben die Methoden ihrer Superklasse(n). Wird eine Methode in der abgeleiteten Klasse neu definiert, spricht man vom Überladen der Funktion. Außerdem können auch MATLAB-Funktionen innerhalb einer Klasse neu definiert werden. Wird eine solche Funktion auf ein Objekt angewendet, erkennt MATLAB dies und verwendet die Klassenmethode statt der MATLAB-Funktion. Darüber hinaus können Operatoren überladen werden. (So kann etwa in einer Polynomklasse der +-Operator als die Addition der einzelnen Elemente der zu addierenden Polynome definiert werden.)

Als Beispiel wird die MATLAB-Funktion `sum`, die auf `Mitarbeiter`-Objekte angewendet einen Fehler hervorruft, überladen, so dass sie die Summe der Gehälter der als Parameter übergegebenen Mitarbeiter zurückgibt:

```
        function summe = sum(varargin) % = beliebig viele Parameter
           summe = 0;
           for i = 1:nargin
              summe = summe + varargin{i}.Gehalt;
           end
        end
```

Die Anwendung sieht wie folgt aus:

```
>> m1 = Mitarbeiter('Max Mustermann',2);   % Gehalt: 2100
>> m2 = Mitarbeiter('Erika Musterfrau',3); % Gehalt: 2400
>> sum(m1,m2)

ans =

        4500
```

Value- und Handle-Klassen

In MATLAB werden zwei Klassentypen unterschieden: Value- und Handle-Klassen.

Value-Klassen sind "normale" Klassen. Objekte solcher Klassen sind mit den darin gespeicherten Daten verknüpft. Wird einem Objekt B der Wert des Objekts A zugewiesen, werden die Daten aus A kopiert und B wird mit dieser Kopie verknüpft. Beispiel:

```
>> a = 2;
>> b = a;
>> b = 5;
>> a

ans =

        2
```

Hier behält `a` den Wert 2, da lediglich die in `b` gespeicherte Kopie von `a` verändert wurde. Sowohl die `Mitarbeiter`-Klasse als auch `double` (der Typ ganz normaler Variablen) sind Value-Klassen.

Im Gegensatz dazu sind Objekte von Handle-Klassen nur Referenzen auf Daten. Wird hier einem Objekt B der Wert des Objekts A zugewiesen, ist B eine zweite Referenz auf den gleichen Daten-

satz, auf den A referenziert. Somit ist eine Veränderung von A auch in B zu sehen. Handle-Klassen erzeugt man, indem man sie als Subklasse der Klasse `handle` definiert:

```
classdef BeispielKlasse < handle
```

Handle-Klassen werden z.B. für Event-Listener oder physische Objekte, die nicht kopiert werden können (wie etwa Drucker) verwendet.

Aufgaben und Übungen

8.1 ☞ *Ballsport:* Erstellen Sie eine Klasse `Spieler` mit den Eigenschaft `Ballbesitz` und den Methoden `anstossen` und `passen`. Die Eigenschaft `Ballbesitz` soll für neue Objekte den Wert 0 haben. Die Methode `anstossen` setzt den Ballbesitz des zugehörigen Spielers auf 1. Die Methode `passen` soll den passenden und einen weiteren Spieler als Parameter erhalten. Sie prüft zunächst, ob der passende Spieler in Ballbesitz ist. Ist dies der Fall, wird `Ballbesitz` für den passenden Spieler auf 0 und für den anderen Spieler auf 1 gesetzt.

8.2 ☞ *Eine Runde Kniffel:* Eine Runde des Spiels Kniffel besteht aus bis zu drei Würfen mit fünf Würfeln. Wenn man im ersten Wurf keine brauchbare Zahlenkombination geworfen hat, darf man beliebig viele Würfel noch einmal werfen. Wenn man immer noch nicht zufrieden ist, darf man ein weiteres Mal beliebig viele Würfel werfen.

Erstellen Sie eine Klasse zur Darstellung einer Runde im Kniffel. Diese Klasse soll die drei möglichen Würfe simulieren können.

a. Erzeugen Sie eine Klasse mit dem Namen *Kniffelrunde* und den Eigenschaften *Wurf* (diese soll später die fünf geworfenen Zahlen enthalten) und *AnzahlWurf* (hierin soll abgespeichert werden, in welchem Wurf sich der Spieler befindet). Initialisieren Sie die Eigenschaft *AnzahlWurf* mit 1.

b. Ergänzen Sie die Klasse um einen Konstruktor. In diesem soll in der Eigenschaft *Wurf* ein 1x5-Vektor aus zufälligen ganzzahligen Zahlenwerten zwischen 1 und 6 abgespeichert werden (Tipp: Verwenden Sie `rand` und `floor`). Sortieren Sie die geworfenen Zahlen der Größe nach (`sort`) und geben Sie sie im Command Window aus.

c. Erstellen Sie die Methode *nochmalWuerfeln*. Diese soll das *Kniffelrunde*-Objekt sowie einen Vektor mit den Indizes der nochmal zu werfenden Würfel als Eingangsgrößen sowie das veränderte Objekt als Ausgangsgröße erhalten. Es soll überprüft werden, ob nicht schon dreimal gewürfelt wurde. Ist dies nicht der Fall, soll die Eigenschaft *AnzahlWurf* um 1 erhöht und ausgegeben werden und die gewünschten Positionen im *Wurf*-Vektor sollen durch neue Zufallszahlen ersetzt werden. Der Vektor soll wieder sortiert und ausgegeben werden.

9 Einführung in Stateflow

Das Paket STATEFLOW® dient zur Modellierung und Simulation ereignisdiskreter Systeme innerhalb von SIMULINK®. Mit Hilfe sogenannter *Zustandsdiagramme* (engl. *state charts*) werden die Modelle graphisch programmiert. Eine Hierarchiebildung ermöglicht eine Aufteilung, welche oft in enger Anlehnung zum realen Prozess gebildet werden kann. Durch eine Übersetzung in eine s-function wird der Rechenzeitbedarf während der Ausführung in Grenzen gehalten.

Es empfiehlt sich, gerade bei umfangreicheren diskreten (Teil-) Systemen, aber auch schon bei relativ kleinen Systemen mit wenigen Zuständen, STATEFLOW® einzusetzen, statt diese Funktionen umständlich mit SIMULINK®-Blöcken zu realisieren. Zu beachten ist allerdings, dass STATEFLOW® ein sehr mächtiges Werkzeug ist: Ein und dasselbe System lässt sich in STATEFLOW® auf sehr unterschiedliche Arten realisieren, wobei selbst darauf zu achten ist, dass die gewählte Realisierung möglichst übersichtlich und nachvollziehbar ist.

9.1 Grundelemente von Stateflow

Die Modellierung der ereignisdiskreten Systeme erfolgt in STATEFLOW® mit *Zustandsübergangsdiagrammen* (engl. *state charts*), welche mit einem graphischen Editor erzeugt werden. Die Charts müssen in ein SIMULINK®-Modell eingebettet sein, welches den Aufruf und den Ablauf des State Charts steuert.

Mit der Befehlseingabe von

```
>> stateflow
```

oder

```
>> sf
```

in der MATLAB-Kommandozeile startet man ein leeres Chart innerhalb eines neuen SIMULINK®-Modells. Alternativ kann man mit Hilfe des Simulink-Library-Browsers aus der Toolbox „Stateflow“ ein leeres Chart durch Ziehen mit der Maus in ein bestehendes SIMULINK®-Modell einfügen.

Hinweis: Zur Ausführung eines Charts wird zwingend ein C-Compiler benötigt. Eventuell muss dieser in MATLAB zuerst mit dem Befehl

```
>> mex -setup
```

konfiguriert werden.

Bedienoberfläche

Durch Doppelklick auf das Chart-Symbol im SIMULINK®-Modell wird der graphische Editor von STATEFLOW® gestartet (siehe Abb. 9.1). Mit der Werkzeugleiste auf der linken Seite können die verschiedenen Chart-Elemente wie Zustände (States) und Transitionen ausgewählt und mit der Maus auf der Arbeitsfläche platziert werden.

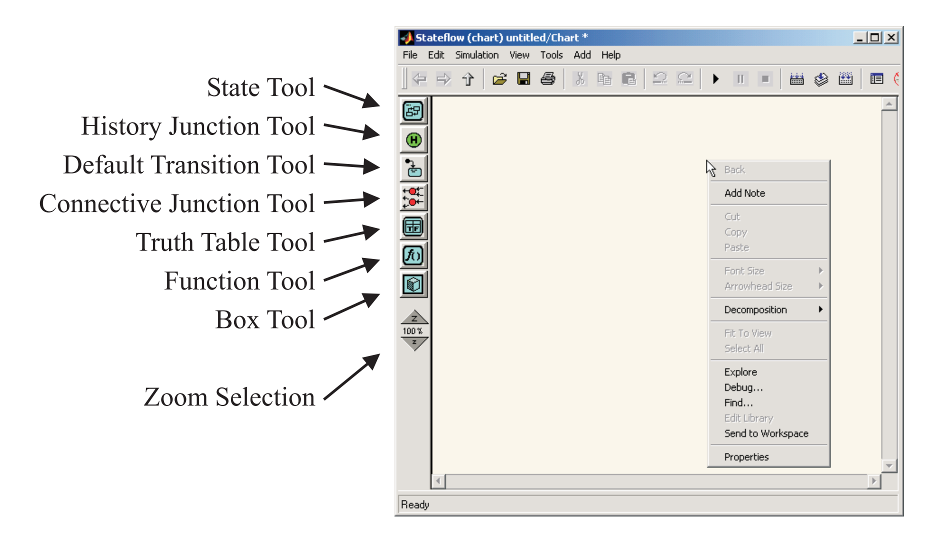


Abbildung 9.1: Graphischer Stateflow Editor

Zustände

Nach Auswahl des State-Tools in der Werkzeugleiste kann durch Klick ein Zustand auf der Arbeitsfläche platziert werden (Abb. 9.2). Zustände können verschoben werden, indem in den leeren Raum im Inneren des Zustandes geklickt und gezogen wird. Innerhalb des Zustandes steht sein Label als Fließtext, welches zwingend vergeben werden muss. Ist noch kein Label eingegeben, erscheint stattdessen ein Fragezeichen.

```
Name1/
entry: aktion1;
during: aktion2;
exit: aktion3;
on event1: aktion4;
```

Abbildung 9.2: Zustand mit komplettem Label

Zum Bearbeiten des Labels klickt man bei markiertem Zustand auf das Fragezeichen bzw. auf das schon vorhandene Label. Das Label (vgl. Abb. 9.2) besteht dabei aus `Name1`, dem Namen des Zustandes, welcher eindeutig (innerhalb einer Hierarchieebene, siehe Abschnitt 9.2) als gültiger C-Variablenname vergeben werden muss. Die weiteren Elemente des Labels sind optional und bezeichnen Aktionen, welche durch diesen Zustand angestoßen werden. Die Syntax für die Festlegung der Aktionen ist die **Action Language**, eine Mischung aus C und der aus MATLAB bekannten Syntax (siehe Abschnitt 9.3). Folgende Schlüsselwörter legen fest, dass die Aktion ausgeführt wird,

`entry`: wenn der Zustand aktiviert wird,

`during`: wenn der Zustand aktiv ist (und das Chart ausgeführt wird),

`exit`: wenn der Zustand verlassen wird,

`on event`: wenn der Zustand aktiv ist und das angegebene Ereignis auftritt.

Transitionen

Eine Transition für einen Zustandsübergang erzeugt man, indem man auf den Rand des Ausgangszustandes klickt und die Maus bis zum Rand des folgenden Zustandes zieht und dort loslässt. Dabei entsteht ein die Transition symbolisierender Pfeil, dessen Form und Lage durch Ziehen verändert werden kann.

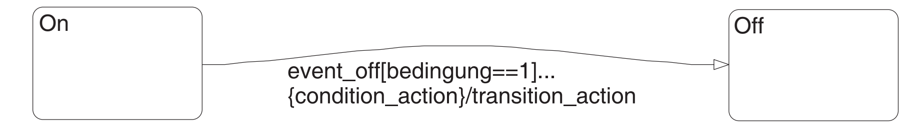


Abbildung 9.3: Transition mit komplettem Label

Ebenso wie der Zustand besitzt die Transition ein Label, welches hier aber nicht zwingend vergeben werden muss. Ein Transitionslabel besitzt die Syntax

```
event[condition]{conditionAction}/transitionAction
```

Alle Komponenten des Labels sind optional und können beliebig kombiniert werden. Eine Transition ist *gültig*, falls

- der Ausgangszustand aktiv ist,
- das Ereignis `event` auftritt oder kein Ereignis angeben ist und
- die angegebene Bedingung `condition` wahr ist oder keine Bedingung gestellt wurde.

Sobald eine Transition gültig ist, wird die `conditionAction` ausgeführt. Die `transitionAction` wird beim Zustandsübergang ausgeführt. Für einfache Transitionen besteht zwischen den beiden Aktionen, abgesehen von der Reihenfolge, kein Unterschied.

Weitere Transitionstypen

Neben den erläuterten einfachen Transitionen gibt es in der Werkzeugleiste weitere Transitionstypen. Die **Standardtransition** (engl. **default transition**, s. Abb. 9.4) legt fest, welcher Zustand bei erstmaliger Ausführung eines Charts aktiv ist. Diese Festlegung ist zwingend erforderlich, weil grundsätzlich zu jedem Zeitpunkt der Ausführung genau ein Zustand aktiv sein muss (innerhalb einer Hierarchieebene). Dazu wird der entsprechende Knopf in der Werkzeugleiste gedrückt und dann der Rand des gewünschten Zustandes angeklickt.

Neben den genannten Transitionstypen gibt es noch die **innere Transition**, welche im Abschnitt 9.2 erläutert wird.

Hinweis: Es können auch mehrere Standardtransitionen innerhalb einer Hierarchieebene verwendet werden. Dabei ist aber mit Ereignissen oder Bedingungen sicherzustellen, dass jeweils genau eine der Standardtransitionen gültig ist.

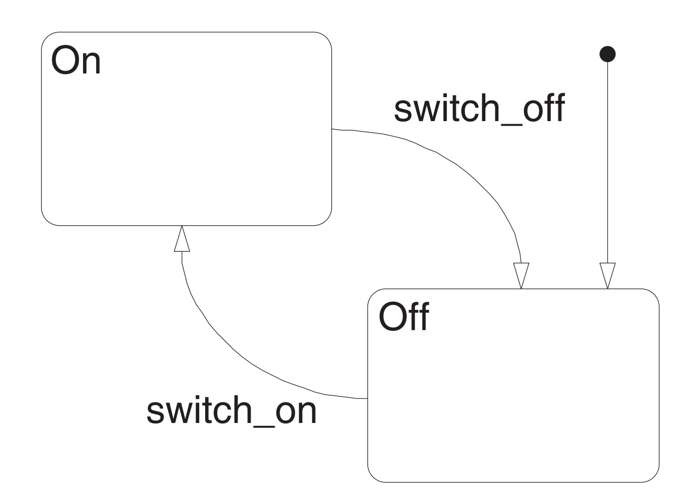


Abbildung 9.4: Chart mit Standardtransition

Verbindungspunkte

Ein weiteres Element ist der **Verbindungspunkt** (engl. **connective junction**). Mit Verbindungspunkten lassen sich komplexere Transitionen durch Zusammenführung und Verzweigung erzeugen. Damit lassen sich Verzweigungen, Fallunterscheidungen, zustandsfreie Flussdiagramme, verschiedene Schleifen, Selbstscheifen u.v.m. erzeugen (siehe Abb. 9.5).

Variablendeklaration

Alle verwendeten Variablen und Ereignisse müssen deklariert werden. Hierzu dient der *Model-Explorer* (siehe Abb. 9.6) im Menü *Tools→Explorer*. Auf der linken Seite des Explorers befindet sich ein hierarchischer Baum, der alle Zustände *aller momentan offenen* State Charts enthält.

Die Deklaration erfolgt in der von objektorientierten Programmiersprachen bekannten Weise. Jedes Objekt kann seine eigenen Deklarationen besitzen. Diese sind dann jedoch nur im jeweiligen Mutterobjekt und den Kindobjekten *sichtbar* (es sei denn, in einem Kindobjekt ist eine Variable gleichen Namens definiert[1]). Auch die *Lebensdauer* der Variablen hängt vom Aktivierungszustand des Mutterobjektes ab. Ist ein Zustand nicht mehr aktiv, sind die in ihm deklarierten Variablen nicht mehr vorhanden („zerstört“).

Variablen und Ereignisse können als *lokal* oder als *Ein-/Ausgänge* von und nach SIMULINK® deklariert werden, wobei letztere nur in der obersten Ebene eines Charts vorkommen können.

Mit dem Menüpunkt „Add“ kann eine Variable (`data`) oder ein Ereignis (`event`) hinzugefügt werden. Bei Doppelklick auf das Symbol links neben dem Namen, öffnet sich ein Dialog, bei dem weitere Optionen festgelegt werden können.

Bei *Ereignissen* lauten die wichtigsten Einstellmöglichkeiten:

`Name`: Name des Ereignisses

`Scope`: Ereignis ist lokal oder Ein-/Ausgang nach Simulink (nur in der Chart-Ebene).

[1] In diesem Fall wird die Variable des Kindobjektes angesprochen. Jedoch führt eine doppelte Vergabe von Variablenbezeichnungen meist zu Unübersichtlichkeiten und sollte generell vermieden werden.

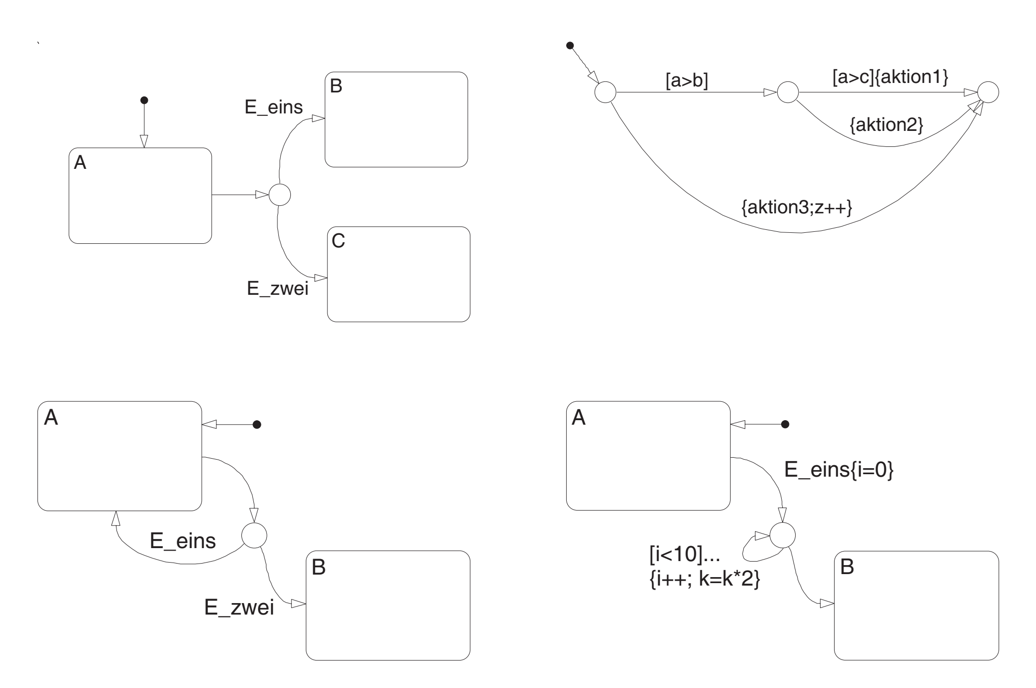


Abbildung 9.5: Komplexe Transitionen durch den Einsatz von Verbindungspunkten, links oben: Verzweigung, rechts oben: Flussdiagramm mit if-Abfrage, links unten: Selbstschleife, rechts unten: for-Schleife

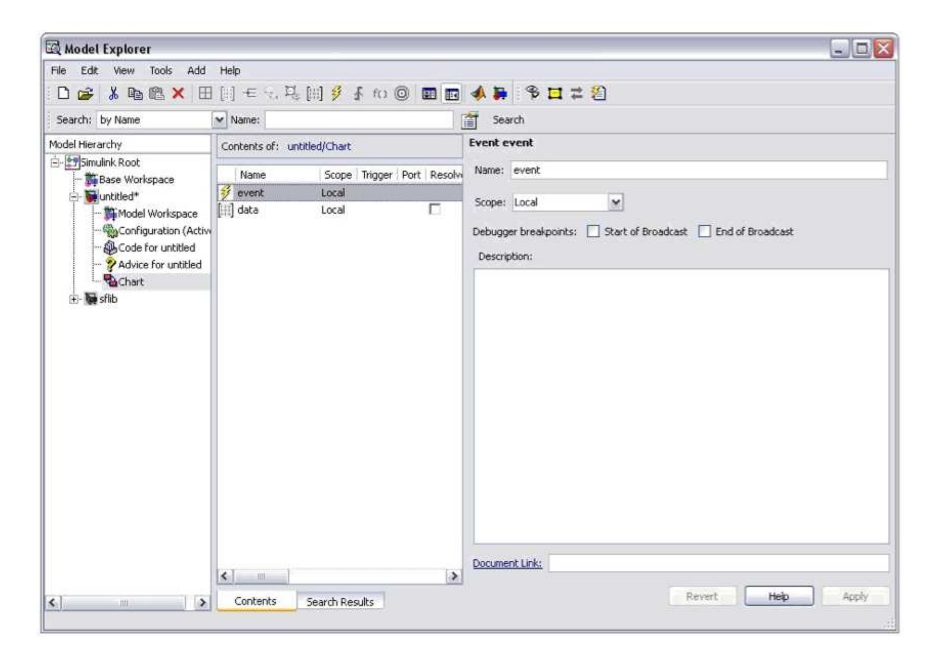


Abbildung 9.6: Der Model-Explorer

`Port`: Nur bei Ausgang: Nummer des Port des Chart-Blockes in Simulink, an dem das Ereignis abgegriffen werden kann.

`Trigger`: Nur bei Ein-/Ausgang: Legt fest, ob eine steigende (von - nach +), eine fallende (von + nach -) oder beide Signalflanken ein Ereignis auslösen. (Wichtig: In jedem Fall muss die Null gekreuzt werden. Ein Übergang von 1 nach 2 löst kein Ereignis aus).
Die Signalflanken werden nur am Anfang eines Simulationsschritts ausgewertet. Mit der Option `Function Call` kann ein Ereignis auch innerhalb eines Simulationsschritts an einen Simulink-Block weitergegeben bzw. empfangen werden.

Bei *Variablen* lauten die wichtigsten Einstellmöglichkeiten:

`Name`: Name der Variablen

`Scope`: Variable ist lokal, konstant oder Ein-/Ausgang nach Simulink (nur in der Chart-Ebene). Eine Konstante kann nicht durch Stateflow-Aktionen verändert werden.

`Port`: Nur bei Ein-/Ausgang: Nummer des Port des Chart-Blockes in Simulink, an dem die Variable anliegt.

`Type`: Typ der Variablen (`double`,`single`,`int32`,...).

`Initialize from`: Die Variable kann durch einen hier eingegebenen Wert („data dictionary“ auswählen und Initialisierungswert ins Feld rechts daneben eintragen) oder aus dem Matlab-Workspace („workspace“ auswählen) zum Simulationsbeginn initialisiert werden.

Werden Ein-/Ausgänge festgelegt, erscheinen diese im Simulink-Modell als Ein- bzw. Ausgangsports beim Chart-Symbol. Zu beachten ist, dass alle *Eingangsereignisse* über den *Triggereingang* als Vektorsignal zugeführt werden müssen (vgl. Abb. 9.7).

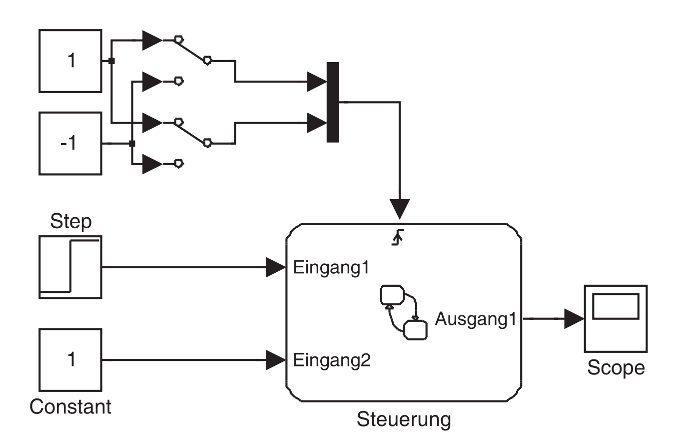


Abbildung 9.7: Chart mit verschiedenen Eingängen: Ereignisse werden als Vektor in den Triggereingang eingespeist, Variablen benutzen Eingangsports.

Neben Variablen und Ereignissen kann auch die *Aktivität eines Zustandes* als Ausgang nach Simulink verwendet werden. Einen solchen Ausgang legt man nicht im Explorer, sondern in der Eigenschaftsdialogbox des Zustandes an (Rechtsklick auf den Zustand, dann `Properties` im Kontextmenü). Nachdem man dort das Kästchen `Output State Activity` markiert, erscheint ein zusätzlicher Ausgangsport beim Chart-Symbol in Simulink und im Stateflow-Explorer. Der Wert dieses Ausgangs beträgt während der Simulation Eins, falls der Zustand aktiviert ist, sonst Null.

Hinweise: Ereignisse können durch die Aktion `/Ereignisname;` (bzw. {`Ereignisname;`} in Zuständen und Transitionen ausgelöst werden.
Die aktuelle Simulationszeit ist Inhalt der Variable „*t*“, welche *nicht* deklariert werden muss.

Charts ausführen

Stateflow-Charts sind in ein Simulink-Modell eingebettet, welches während der Simulationszeit den Aufruf des Charts auslöst. In der Eigenschaftsdialogbox des Charts (Menü *File*→*Chart Properties*) ist neben weiteren Optionen festzulegen, unter welchen Bedingungen der Chart aufgerufen wird. Diese Eigenschaft wird unter dem Punkt `Update method` eingestellt (vgl. Abb. 9.8).

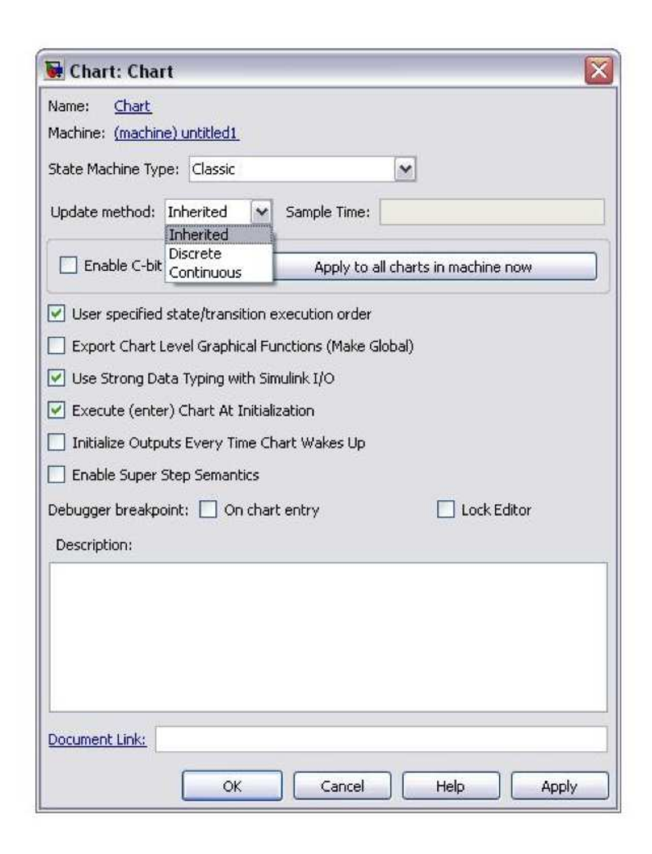


Abbildung 9.8: Einstellung der Update-Methode in den Chart Properties

Der Chart wird ausgeführt,

`Inherited`: wenn durch die vorangehenden Blöcke eine Eingangsvariable neu berechnet wird (falls keine Eingangsereignisse deklariert sind),

`Discrete`: in festen Zeitabständen (unabhängig vom Rest des Simulinkmodells). Dieser Zeitabstand wird im Feld `Sample Time` festgelegt.

`Continuous`: bei jedem Integrationsschritt (auch bei Minor Steps).

Ein Einschalten der Option `Execute (enter) Chart At Initialization` bewirkt, dass beim Simulationsstart das Chart einmal ausgeführt wird, unabhängig von der gewählten Aktualisierungsmethode.

Die Simulation kann wie üblich in Simulink oder im Stateflow-Editor durch Drücken des Play-Buttons oder im Menü `Simulation` gestartet werden. Bei erstmaliger Simulation oder nach Veränderungen des Charts wird zuerst das Chart durch Kompilieren in eine s-function umgewandelt. Dieser Schritt kann relativ zeitaufwendig sein und zu einer merklichen Verzögerung führen, bis die Simulation abläuft.

Ist der Stateflow-Editor während der Simulationszeit offen, wird der Ablauf der Zustandsaktivierungen graphisch animiert. Zusätzliche Verzögerungen zur besseren Sichtbarkeit können im Debugger (Menü `Tools - Debugger`) konfiguriert werden. Hier kann die Animation auch gänzlich abgeschaltet werden. Viele weitere Funktionen unterstützen dort den Anwender bei der Fehlersuche. So können *Breakpoints* an verschiedenen Stellen gesetzt werden, automatisch Fehler (Zustandsinkonsistenzen, Konflikte, Mehrdeutigkeiten, Gültigkeitsbereiche, Zyklen) gesucht werden, zur Laufzeit Werte von Variablen und aktive Zustände angezeigt werden.

Hinweis: Die übergeordneten Simulationseigenschaften (Simulationsdauer, Integrationsschrittweiten, etc.) werden durch das Simulink-Modell festgelegt.

Aufgaben

9.1 ☞ *Erste Schritte:* Sie kennen nun alle Elemente, um einfache Zustandsautomaten zu simulieren. Experimentieren sie selbst: Erstellen Sie einfache Charts und lassen diese ablaufen. Wenn nötig, orientieren sie sich an den bisher abgebildeten Beispielen.

9.2 ☞ *R/S FlipFlop:* Erstellen Sie das Modell eines R/S-Flipflops (ein FlipFlop stellt ein 1-Bit Speicherelement dar). Dieses verfügt über einen Ereigniseingang `clk`, über den die Taktung erfolgt, zwei Dateneingänge `R` und `S`, die bei jeder steigenden Taktflanke ausgewertet werden und einen Datenausgang `Q`, der den aktuellen Speicherzustand (0 oder 1) des FlipFlops repräsentiert. Die Schaltsemantik findet sich in der folgenden Tabelle wieder:

Clk	R	S	Q
1 (rising edge)	0	0	unverändert
1 (rising edge)	1	0	0 (zurücksetzen)
1 (rising edge)	1	0	1 (setzen)
1 (rising edge)	1	1	nicht zulässig, hier: unverändert
0	-	-	unverändert (- don't care)

Erproben Sie das Modell in einer geeigneten Simulink-Umgebung, die den Speicherzustand des FlipFlop ausgibt, ein einfaches Umschalten der Eingänge ermöglicht und eine periodische Taktung vornimmt.

9.3 ☞ *Aufzugsteuerung:* Erstellen Sie ein diskretes Modell für einen Aufzug über vier Stockwerke (UG, EG, Stock1, Stock2). Sehen Sie für jedes Stockwerk einen entsprechend benannten Zustand vor. Die Steuerung des Aufzuges soll zunächst vereinfacht erfolgen: Erzeugen Sie mit Hilfe von Schaltern in Simulink die Ereignisse `Rauf` und `Runter`, welche einen entsprechenden Zustandswechsel bewirken. Die aktuelle Position des Aufzuges soll an Simulink übergeben werden.
Testen Sie anschließend das Modell.

9.4 ☞ *Dreitank:* Für den in Abb. 9.9 dargestellten Dreitank sollen die Füllhöhen in den drei Flüssigkeitsbehältern zur Visualisierung an einem Steuerpult durch jeweils drei Lampen dargestellt

werden, welche die Zustände `Leer`, `Normal` und `Überlauf` anzeigen. Erzeugen Sie das Zustandsdiagramm für Tank 1. Beschriften Sie die Transitionen mit Kurznamen für die möglichen Ereignisse `Tank 1 beginnt überzulaufen`, `Tank hört auf überzulaufen`, etc. (Die Beschriftungen dürfen jedoch keine Leerzeichen oder Umlaute enthalten.) Erzeugen Sie diese Ereignisse über zu definierende Triggereingänge zu Stateflow, die Sie - während die Simulation läuft - testweise unter Simulink triggern.

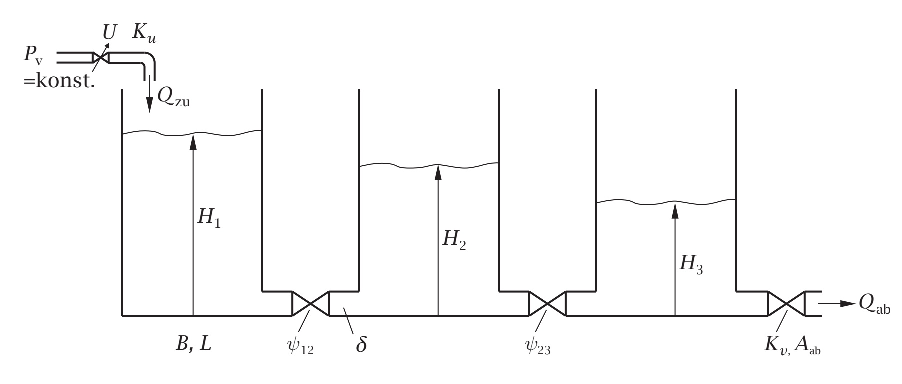


Abbildung 9.9: Dreitank

9.2 Weitere Strukturen und Funktionen

Neben den bisher genannten Basiselementen gibt es weitere Strukturen, welche komplexere Funktionen, aber auch die einfachere und übersichtlichere Darstellung von ereignisdiskreten Systemen ermöglichen.

Superstates und Subcharts

Mit *Superstates* lassen sich zusammengehörige Zustände zu einem Oberzustand zusammenfassen. Dies geschieht dadurch, dass zuerst mit dem State-Tool ein neuer Zustand im Stateflow-Editor angelegt wird. Dann wird durch Klicken und Ziehen an den Ecken des Zustandes (der Mauszeiger verwandelt sich in einen Doppelpfeil) der Zustand soweit vergrössert, bis er alle gewünschten Unterzustände umfasst (siehe Abb. 9.10).

Für das Innere eines solchen Superstates gelten dieselben Regeln wie für die oberste Ebene (Initialisierung, Anzahl aktiver Zustände etc.). Ein Superstate kann wie ein herkömmlicher Zustand Ziel und Ausgangspunkt von Transitionen sein. Er wird dann aktiviert, wenn er selbst oder einer seiner Unterzustände das Ziel einer gültigen Transition ist. Ist ein Superstate nicht aktiviert, ist auch keiner seiner Unterzustände aktiv. Die Unterzustände können dabei als eigenständiges Chart innerhalb des Superstates aufgefasst werden.

Die Aufteilung in Superstates und Unterzustände kann jederzeit durch Änderung der Größe und Lage der Zustände verändert werden. Um eine unabsichtliche Veränderung zu verhindern, kann der Inhalt eines Zustandes *gruppiert* werden. Dazu doppelklickt man entweder in das Innere eines Zustandes oder wählt im Kontextmenü (Rechtsklick) des Zustandes den Punkt `Make Contents - Grouped` aus. Der Zustand wird dann grau eingefärbt.

Bei gruppierten Zuständen bleiben alle Unterzustände und Transitionen sichtbar, können aber

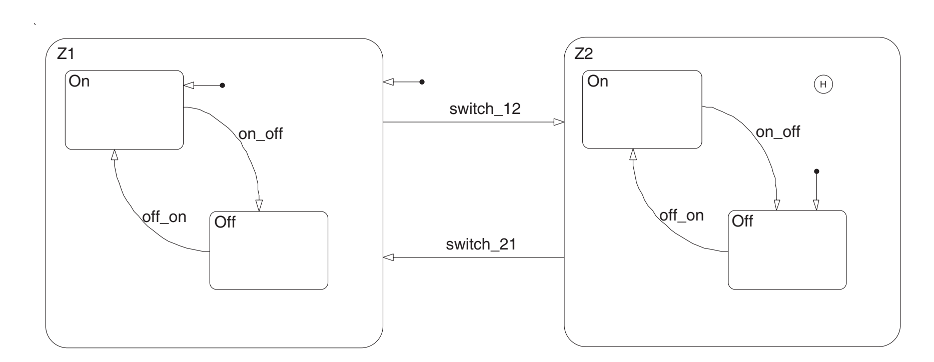


Abbildung 9.10: Chart mit zwei Superstates. Der rechte Superstate Z2 besitzt eine History Junction.

nicht mehr bearbeitet werden. Wird im Kontextmenü der Punkt `Make Contents - Subcharted` ausgewählt, wird der Inhalt verdeckt. Damit lassen sich Charts mit komplizierten Teilfunktionen übersichtlicher darstellen. Die Funktion bleibt allerdings gegenüber gruppierten Superstates dieselbe. Durch Doppelklick kann ein Subchart geöffnet und dann bearbeitet werden. Zurück in die höhere Hierarchiebene kommt man mit dem „Nach-Oben-Pfeil“ in der Symbolleiste.

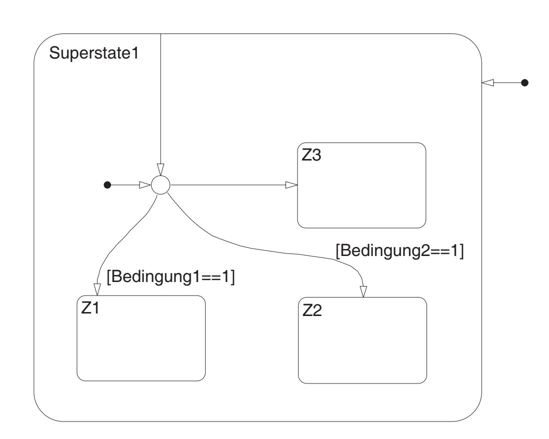


Abbildung 9.11: Superstate mit einer Inneren Transition

Verliert ein Superstate durch eine Transition seine Aktivität, werden auch alle Unterzustände passiv. Bei einer neuerlichen Aktivierung des Superstate entscheidet dann wieder die Standardtransition, welcher Unterzustand die Aktivierung erhält. Dies kann man dadurch verhindern, indem man in den Zustand aus der Werkzeugleiste eine *History Junction* setzt. Nun erhält derjenige Unterzustand, welcher vor dem Verlust aktiviert war, die Aktivierung zurück (vgl. Abb. 9.10).

In einem Superstate kann eine weiterer Transitionstyp eingesetzt werden, die *Innere Transition*. Sie wird erzeugt, in dem eine Transition vom Rand des Superstates zu einem seiner Unterobjekte mit der Maus gezogen wird. Diese Transition wird immer auf Gültigkeit überprüft solange der Superstate aktiv ist, unabhängig von seinen Unterzuständen. Ein Beispiel für die Verwendung einer Inneren Transition ist die Vereinigung einer verzweigten Zustandskette. Anstatt Transitionen von allen Enden dieser Kette einzuzeichnen, genügt oft der Einsatz einer Inneren Transition (vgl. Abb. 9.11).

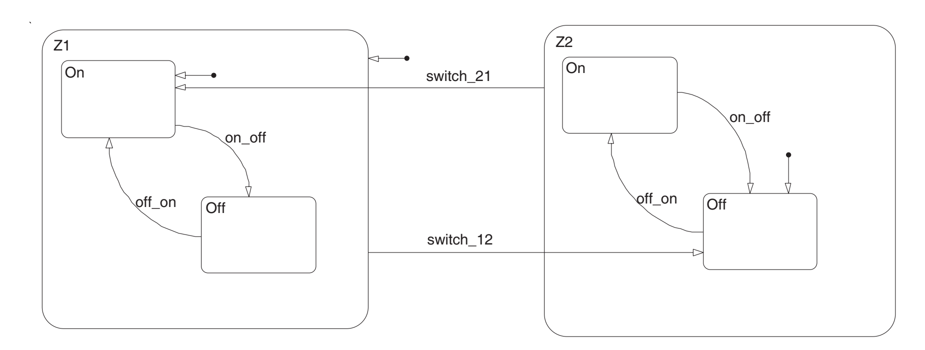


Abbildung 9.12: Zustandsdiagramm mit hierarchieübergreifenden Transitionen

Hierarchieübergreifende Transitionen und Wurmlöcher

Transitionen können auch übergreifend über alle Hierarchien angelegt werden. So sind alle Verbindungen zwischen Superstates und Unterzuständen möglich. (siehe Abb. 9.12). Jedoch bleibt die Regel gültig, dass bei Zuständen in exklusiver Oder-Anordnung immer genau ein Zustand pro Superstate aktiv ist.

Hat man einen Zustand als Subchart anlegt, ist dessen Inhalt verdeckt. Mit Hilfe eines *Wurmloches* können auch Transitionen von und zu den Inhalten eines Subchartes angelegt werden. Zieht man eine Transition auf die Fläche eines Subchartes, wird in der Mitte ein Wurmloch angezeigt. Bewegt man sich mit der Maus bei gedrückter Maustaste auf dieses Wurmloch, wird der Subchart geöffnet und man kann die Transition dort anbringen. Auch in umgekehrter Richtung ist der Vorgang möglich. Hierzu zieht man eine Transition vom Inneren eines Subcharts auf dessen Äußeres, so dass dort ein Wurmloch sichtbar wird. Ein Beispiel solcher Transitionen ist in Abb. 9.13 dargestellt.

Transitionsprioritäten

Je nach Beschriftung der Transitionen durch die Labels kann es zu Fällen kommen, bei denen zwei unterschiedliche Transitionen prinzipiell gleichzeitig gültig werden. Solche Mehrdeutigkeiten sollte man zwar bei der Charterstellung unbedingt vermeiden, jedoch gibt es für eine Auflösung dieses Konflikts feste Regeln.

So wird diejenige Transition ausgeführt, welche

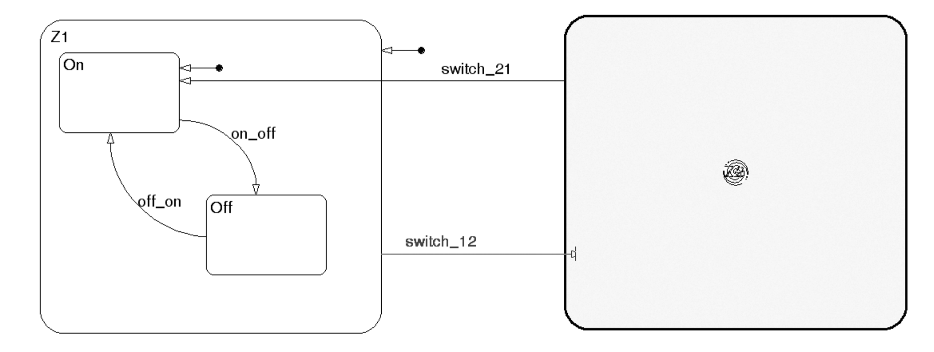


Abbildung 9.13: Zustandsdiagramm mit Transitionen, welche in ein Subchart hineinführen.

- zu einem Zielzustand höherer Hierarchie führt,
- ein Label mit Ereignis- und Bedingungsabfrage besitzt,
- ein Label mit Ereignisabfrage besitzt,
- ein Label mit Bedingungsabfrage besitzt,

Ist mit diesen Kriterien noch keine eindeutige Entscheidung möglich, wird die Reihenfolge graphisch bestimmt: Es wird diejenige Transition zuerst ausgeführt, deren Abgangspunkt der linken oberen Ecke eines Zustandes (bzw. der 12-Uhr-Stelle eines Verbindungspunktes) im Uhrzeigersinn als nächster folgt.



Abbildung 9.14: Chart mit parallelen Zuständen

Parallele Zustände

Bisher bezogen sich alle Ausführungen auf die exklusive Anordnung (Oder-Anordnung) von Zuständen, bei der immer genau ein Zustand aktiviert ist. Neben dieser Anordnung gibt es die *parallele Und-Anordnung*, bei der alle Zustände einer Hierarchieebene gleichzeitig aktiv sind und nacheinander abgearbeitet werden.

Innerhalb jeder Hierarchieebene (Chart, Superstate) kann die Art der Anordnung getrennt festgelegt werden. Dies geschieht im Kontextmenü eines zugehörigen Zustandes unter `Decomposition`. Wird auf `Parallel (AND)` umgeschaltet, ändert sich die Umrandung der betroffenen Zustände zu einer gestrichelten Linie und es wird eine Zahl in der rechten oberen Ecke eingeblendet (siehe Abb. 9.14). Diese Zahl gibt die Ausführungsreihenfolge der Zustände an. Die parallelen Zustände werden nacheinander von oben nach unten und von links nach rechts ausgeführt.

Mit parallelen Zuständen können Systeme modelliert werden, welche parallel ablaufende Teilprozesse besitzen.

Boxen

Ein weiteres Gruppierungsmittel stellen *Boxen* dar. Eine Box kann mit dem entsprechenden Werkzeug der Werkzeugleiste erzeugt werden. Auch kann ein Zustand mit seinem Kontextmenü mit dem Punkt `Type` in eine Box umgewandelt werden (und umgekehrt).

Mit einer Box können Bereiche eines Charts zur besseren Übersichtlichkeit eingerahmt werden (siehe Abb. 9.15). Zu beachten ist, dass die Box die Ausführungsreihenfolge verändern kann, sonst jedoch keinen Einfluss auf die Funktion eines Charts besitzt.

Auch Boxen können in der bekannten Weise durch den Eintrag `Make Contents` des Kontextmenüs gruppiert und zu einem Subchart umgewandelt werden, so dass Inhalte nicht mehr veränderbar bzw. sichtbar sind.

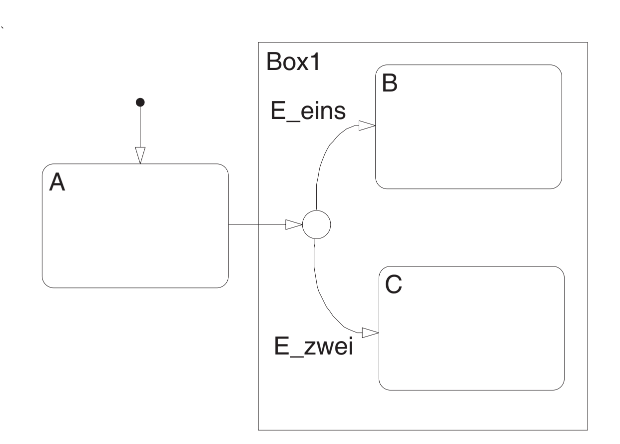


Abbildung 9.15: Zustandsdiagramm mit Box

Weitere graphische Elemente

Wahrheitstabellen

Wahrheitstabellen (Truth Tables) in Stateflow bestehen aus Bedingungen (conditions), Entscheidungen (descisions) und Aktionen (actions) (Abb. 9.16). Jede Bedingung muss entweder als wahr (nicht Null) oder als falsch (Null) ausgewertet werden können. Durch die Entscheidungen wird angegeben, ob die zugerhörige Bedingung erfüllt oder nicht erfüllt sein muss. Es wird dann entsprechend die Aktion der Spalte ausgeführt, deren Entscheidungen alle zutreffen.

Condition	Decision 1	Decision 2	Decision 3	Default Decision
x == 1	T	F	F	-
y == 1	F	T	F	-
z == 1	F	F	T	-
Action	t = 1	t = 2	t = 3	t = 4

Abbildung 9.16: Wahrheitstabelle

Embedded Matlab Functions

Ähnlich wie in Simulink können auch in Stateflow Embedded Matlab Functions verwendet werden. Diese Funktions ist nützlich. um Algorithmen zu programmieren, die sich leichter in der Matlab Sprache beschreiben lassen, als grafisch in Stateflow.

Simulink Functions

In Stateflow können Simulinkt Funktionen eingebunden werden. Eine Simulink Funktion ist ein Block, in dem Simulink Modelle erstellt werden können. Eine typische Anwendung dafür ist es Lookup Tables in ein Stateflow Modell zu integrieren.

Aufgaben

9.5 ☞ *Paralleler Fertigungsprozess:* Modellieren Sie einen Fertigungsprozess bei dem ein Bauteil aus zwei Baugruppen (B1 und B2) hergestellt wird. Die Fertigstellung einer Baugruppe soll über Simulink getriggert werden und so ein Ereignis auslösen. Erst wenn beide Baugruppen vorliegen, kann der Schritt `Zusammensetzen` durchgeführt werden. Nach Fertigstellung des Bauteils beginnt der Prozess von vorne. Testen Sie anschließend das Modell in einer geeigneten Simulink-Umgebung, in der über Schalter die Fertigstellung der Baugruppen an das Stateflow-Modell propagiert werden können.
Tipp: Verwenden Sie bei der Modellierung einen Takteingang (Ereigniseingang), der das Modell periodisch aktiviert.

9.6 ☞ *Erweiterung der Aufzugsteuerung:* Ergänzen Sie das Modell des Aufzuges um folgende Funktionen:

- Integrieren Sie einen Hauptschalter, mit dem der Aufzug ein- bzw. ausgeschaltet wird.
- Erweitern Sie die Steuerung nun derart, dass von Simulink ein Ziel-Stockwerk vorgegeben wird.
- Berücksichtigen Sie den Zustand der Türe (offen - geschlossen). Nach Ankunft des Aufzuges öffnet die Türe für eine gewisse Zeitspanne und schliesst danach wieder. Stellen Sie sicher, dass sich der Aufzug nur bei geschlossenen Türen bewegt.

Tipps: Verwenden Sie auf der obersten Ebene zwei parallele Zustände. Steuern Sie nun die `Rauf`- und `Runter`-Ereignisse geeignet intern. Betten Sie die bisherigen Stockwerks-Zustände in einen Zustand `Tür zu` ein.

9.3 Action Language

In diesem Abschnitt werden die verschiedenen Elemente der Action Language kurz erläutert, welche in den Labels der Zustände und Transitionen benötigt werden.

Besondere Bedeutung kommt den in Tab. 9.1 genannten Schlüsselwörtern zu, welche nicht als Variablen- oder Ereignisnamen verwendet werden können.

In den Tabellen 9.2 bis 9.4 sind die in der Action Language definierten Operatoren zusammengestellt. Alle Operationen gelten dabei nur für *skalare* Größen. Ausnahmen bilden einige wenige Matrizenoperatoren, welche in Tabelle 9.5 zusammengestellt sind. Zu beachten ist, dass der Zugriff auf Elemente von Matrizen in anderer Weise erfolgt als in Matlab!

Wird auf eine Variable oder ein Ereignis zugegriffen, versucht Stateflow diese in derselben Hierarchieebene zu finden, in der die aufrufende Aktion steht. Ist die Suche nicht erfolgreich, wird

Schlüsselwort **Abkürzung**	**Bedeutung**
`change(data_name)` `chg(data_name)`	Erzeugt ein lokales Event, wenn sich der Wert von `data_name` ändert.
`during` `du`	Darauf folgende Aktionen werden als *During Action* eines Zustandes ausgeführt.
`entry` `en`	Darauf folgende Aktionen werden als *Entry Action* eines Zustandes ausgeführt.
`entry(state_name)` `en(state_name)`	Erzeugt ein lokales Event, wenn der angegebene Zustand aktiviert wird.
`exit` `ex`	Darauf folgende Aktionen werden als *Exit Action* eines Zustandes ausgeführt.
`exit(state_name)` `ex(state_name)`	Erzeugt ein lokales Event, wenn der angegebene Zustand verlassen wird.
`in(state_name)`	Bedingung, welche als `true` ausgewertet wird, wenn der angegebene Zustand aktiv ist.
`on event_name`	Darauf folgende Aktionen werden ausgeführt, wenn das angegebene Ereignis auftritt.
`send(event_name,state_name)`	Sendet das angegebene Ereignis an den Zustand `state_name`.
`matlab(funktion,arg1,arg2,...)` `ml(funktion,arg1,arg2,...)`	Aufruf der angegebenen Matlab-Funktion mit den Argumenten `arg1`, `arg2`,...
`matlab.var` `ml.var`	Zugriff auf die Variable `var` des Matlab-Workspaces.

Tabelle 9.1: Schlüsselwörter der Action Language

schrittweise in den darüberliegenden Hierarchieebenen weitergesucht. Soll auf ein Objekt eines anderen Mutter-Objekts zugegriffen werden, so muss der volle Pfad dorthin angegeben werden (Beispiel: `Superstate2.Zustand5.Anzahl`).

Operator	**Beschreibung**
`a + b`	Addition
`a - b`	Subtraktion
`a * b`	Multiplikation
`a / b`	Division
`a %% b`	Restwert Division (Modulus)

Tabelle 9.2: Numerische Operatoren

<table>
<tr><th>Operator</th><th>Beschreibung</th></tr>
<tr><td>a == b</td><td>Gleichheit</td></tr>
<tr><td>a ~= b</td><td>Ungleichheit</td></tr>
<tr><td>a != b</td><td></td></tr>
<tr><td>a > b</td><td>Größer-Vergleich</td></tr>
<tr><td>a < b</td><td>Kleiner-Vergleich</td></tr>
<tr><td>a >= b</td><td>Größer-Gleich-Vergleich</td></tr>
<tr><td>a <= b</td><td>Kleiner-Gleich-Vergleich</td></tr>
<tr><td>a && b</td><td>Logisches UND</td></tr>
<tr><td>a & b</td><td>Bitweises UND</td></tr>
<tr><td>a || b</td><td>Logisches ODER</td></tr>
<tr><td>a | b</td><td>Bitweises ODER</td></tr>
<tr><td>a ^ b</td><td>Bitweises XOR</td></tr>
</table>

Tabelle 9.3: Logische Operatoren

Mit den temporalen Logikoperatoren in Tabelle 9.6 lassen sich Bedingung an ein mehrfaches Auftreten von Ereignissen knüpfen. Sie dürfen nur in Transitionen mit einem Zustand als Quelle und in Zustandsaktionen verwendet werden. Das Aufsummieren von Ereignissen erfolgt nur, solange der Quellzustand aktiv ist. Bei einem Zustandswechsel wird der Zähler auf Null zurückgesetzt.

Operator	Beschreibung
`~a`	Bitweise Invertierung
`!a`	Logische NOT Operation
`-a`	Multiplikation mit -1
`a++`	Variable um 1 erhöhen
`a--`	Variable um 1 erniedrigen
`a = expression`	Zuweisung an die Variable
`a += expression`	identisch mit `a = a + expression`
`a -= expression`	identisch mit `a = a - expression`
`a *= expression`	identisch mit `a = a * expression`
`a /= expression`	identisch mit `a = a / expression`

Tabelle 9.4: Unäre Operatoren und Zuweisungen

Operator	Beschreibung
`matrix1 + matrix2`	Elementweise Addition zweier gleichgroßer Matrizen
`matrix = n`	Belegung aller Matrixelemente mit dem Skalar `n`
`matrix * n`	Multiplikation aller Matrixelemente mit dem Skalar `n`
`matrix[i][j]`	Zugriff auf das Element (`i`,`j`) der Matrix

Tabelle 9.5: Matrizenoperatoren

Operator	Beschreibung
`after(n,event)`	Wahr, wenn das Ereignis mindestens `n`-mal aufgetreten ist.
`before(n,event)`	Wahr, wenn das Ereignis weniger als `n`-mal aufgetreten ist.
`at(n,event)`	Wahr, wenn das Ereignis genau `n`-mal aufgetreten ist.
`every(n,event)`	Wahr, wenn das Ereignis genau `n`-mal, 2`n`-mal,... aufgetreten ist.

Tabelle 9.6: Temporale Logikoperatoren